



Python Web

开发实战

董伟明 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书按照一个 Web 产品从无到有、从简单变复杂、从基础到进阶的过程，多角度、全方位讲述了 Python Web 开发。内容涉及 Web 框架、测试、数据库、消息队列、服务化、持续集成等，把网站工程的全貌展现在读者的眼前，从其中可以了解 Web 工程从开发到上线的完整流程。另外，作者对当前现在正在流行的技术或工具，如 Flask、Celery、Jupyter、Supervisor、SaltStack、Pandas 等都有较为详细的阐述，可作为技术选型时的参考。

对于 Web 开发者、使用 Python 语言的运维工程师和运维开发工程师、想提高 Python 技能的开发者、想了解 Python Web 开发的其他开发者，本书都适合阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Python Web 开发实战 / 董伟明著. —北京：电子工业出版社，2016.9

ISBN 978-7-121-29733-5

I. ① P…II. ① 董…III. ① 程序开发工具—程序设计 IV. ① TP311.561

中国版本图书馆 CIP 数据核字（2016）第 200131 号

责任编辑：许 艳

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：31.5 字数：616.9 千字

版 次：2016 年 9 月第 1 版

印 次：2016 年 9 月第 1 次印刷

定 价：105.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819 faq@phei.com.cn。

推荐序一

作为一名使用 Python 作为主力开发语言十多年的老码农，常常被人要求推荐 Python 相关的图书。经常推荐的都是一些讲解 Python 语言本身的图书，而专注在 Python 的常见应用领域——Web 开发上的好书，却一直是凤毛麟角。也曾有出版社的朋友约我写一本，但我畏惧写书的艰辛，一直不敢答应。得知伟明的《Python Web 开发实战》一书即将出版，欣慰异常，心想伟明写这个专题实在是再合适不过，必为佳作。读完书稿，果不其然。

由于 Python 具有开发快速、适合多人协作、库丰富、社区成熟等优点，因此是一门非常适合用于 Web 开发的语言。国外的 Youtube、Quora，国内的豆瓣、知乎等，均是以 Python 为主要语言开发的。说起 Python 的 Web 开发，很多人会理解成只要学会某个 Web 框架，能写代码查查数据库，写模板渲染出 HTML，最多再学一下配个 Web Server 把服务启动起来就行，没什么内容。多数 Python 书里“Web 开发”的章节一般也就是讲这些。但其实要完成生产可用的，能够应对一定规模访问量的 Web 系统，Web 开发工程师要学会的远远不止这些。环境搭建、API 设计、网站架构、系统管理、持续集成、服务化、数据处理、并发处理等，这些都是必要的，而且需要付出大量的努力才能掌握的知识。

伟明一直对技术抱有极强的兴趣，也有着优秀的动手能力。我对他的第一印象是从他发给豆瓣的求职信开始的：“目前我给 glances、Salt、tmux-powerline、supervisor、mongo-python-driver、circus、graphite-web、Diamond、autopep8、graph-explorer、pip、Celery 等开源项目贡献过代码，给 Python 标准库 logging 贡献过代码。”能够参与到这么多一线开源软件中的应聘者，确实少见。入职豆瓣后，伟明也表现出了对 Web 开发的深刻理解，很快成为豆瓣多个 Web 产品的主力，并几乎凭一人之力完成了 CODE 项目的私有依赖剥离和开源。

伟明把他个人多年 Web 开发的经验，以及豆瓣十年来数百名优秀工程师在 Web 开发上最佳实践的积累，凝聚在了《Python Web 开发实战》这本书里，多维度、全面地介绍了 Python Web 开发涉及的各种技术。更难能可贵的是，他还在这本书里留下了非常多关于这些技术

的思索：为什么要使用某个技术？某个需求都有哪些技术可以满足？如何取舍？这种不仅要知其然，还要知其所以然的态度，是工程师快速成长必备的。

这样的一本书能够出版，对于国内的 Python 开发者绝对是个福音。我向所有有兴趣使用 Python 做 Web 开发的开发者们，强烈推荐此书。

洪强宁

爱因互动 CTO

前豆瓣首席架构师

前宜信大数据创新中心首席架构师

推荐序二

这篇序酝酿了好几天，今天终于动笔写下了第一个字。说实话，很久没有看到关于 Python 的好书了，尤其是国人自己的原创书。Web 开发本身就是一件很庞杂的事情，模板渲染、API 的开发、后端的部署，能在一本书中把这些问题都说清楚并不容易。作者伟明与我都出身豆瓣，虽然没有同时期在豆瓣工作过，但豆瓣的 CODE 系统把我们俩联系到了一起。他是 CODE 的三代目，通过 CODE 里一行行 Python 代码，仿佛知道了彼此的心意。感谢伟明把豆瓣的一些工程实践进行了整理和总结，这是本书最宝贵的一点。而本书的精华在于他对各种技术使用场景的探讨：那些库谁都会用，但在什么场景使用，在生产环境中这个库的表现到底如何，则不见得有认真的思考。期望将来还可以看到越来越多这样的图书，祝此书大卖。

清风

SAY CEO

前豆瓣技术总监

推荐序三

一次真诚的倾诉

自从 CPyUG 列表订阅人数超过 10,000 以后，我就认为中文的 Python 学习资料足够多了，而最正确的自学姿势应该从官方文档开始。但是，《Python Web 开发实战》一书又改变了我这一偏见。

有道是：“出版是最好的记忆”，伟明亲身证实了这一点。作为一名普通的程序员，只从一个念头出发，独自写出了 500 多页的技术图书，这实在是一件令人敬佩的事。凡是写过书，特别是写过技术图书的人都知道——写书，难的不是写什么，如何写，而是要耐得住寂寞！

在中国生活原本就如此艰辛，无论上学还是工作，周围永远有无数同侪在竞争；而写书几乎是负收入的经济行为，特别是技术图书。当然，图书印刷出来，奉上对家人的感谢，是一种别样的程序员式的浪漫情怀，这种成就感不足为外人道矣。

伟明打动我，让我为他写推荐序，就在于他说自己写书的那个简单的初衷：让公司所有人都知道 Web 开发究竟是什么样的，从而能更好地协同。这其实已经是佛陀流传经文普度众生的大情怀了！

Python 是如此活跃的一种语言，几乎伴随互联网的发生而创立，又伴随互联网的极速发展而繁荣，在 Web 开发领域拥有全栈式的技术生态；又因为脚本语言以及其自身语言的人性化设计，通过 Web 勾联到了几乎所有计算机应用领域，这也导致在特定问题上，Python 总是有一堆解决方案可以选择，而不像其他语言，只有一种方案。但是选择过多，其实也导致了学习成本的增加。

伟明将自身在 Web 领域所有方面的经验提炼后整理成书，本质上是将几十个关联产品的官方文档，结合具体工程经验进行了梳理，给出了领域问题最佳方案的关键思考点和自己的

答案，而更加可贵的是，给出了这些思考点的来源，以及形成过程，即给出了解决各种 Web 领域问题的思维模式。

从前后几个版本的书稿也可以看出，如果没有这本图书的艰苦整理，伟明自己也难以形成这种宏观 + 微观能同时作用的思考模式。所以，我一直认为：“输出是更加残酷的输入。”要将纷繁零散的经验，变成他人可以习得的技能，要组织成叙述合理、案例得当、结构清晰的图书，这个过程本身就得分对自己的所有积累进行反复的再学习、解析和增补。其中的工作量远远不止这几百页书的内容。

更加奇妙的是，在没有这部书稿前，其实我们并不熟悉，只是在社区列表中见过邮箱名而已。但是，有了独有的知识成果后，伟明就有了立场，也有了动机和理由，邀请我以及类似洪教授/Limodou 这些中国 Python 学习者的前辈来评点和审核书稿，获得直接的联系，即人脉。

所以，我在郑重推荐此书的内容之外，更加倡议大家向伟明学习——敢于写书，通过真诚的技术图书总结自己的过去，获得更好的未来，帮助更多的 Pythonista。

Zoom.Quiet（大妈）

优视眼动科技 CTO

Python 中文社区创始人之一及管理员

OBP 及蟒营工程设计者兼主持人

Zoom.Quiet（大妈）

从 2002 年开始接触 Python，积极推广 Pythonic，筹办了自 2012 年起连续四届 PyCon 中国大会，编撰有《可爱的 Python》等图书。作为大家熟知的社区“大妈”，主持了 OSTC 2015“程序媛专场”，坐实了这一称号，得到广大程序员认可。

推荐序四

说起来给《Python Web 开发实战》一书写序还真是很突然。2016 年 5 月 30 日，我突然被拉到了一个微信群里，正觉得纳闷的时候，看到群里 Zoom.Quiet 的介绍，才知道是怎么回事。原来《Python Web 开发实战》已经基本成书，让大家看一看。对于本书的作者董伟明，我们没有在线下交流过，但是对 Python 的热爱时不时地会把大家通过某种方式吸引到一起。

这是一本原创图书，从书名来看是和 Web 相关的，而 Web 领域正好和我的兴趣以及平时的工作相关。作为一个开源 Web 框架的开发者，自然对 Web 开发的内容比较感兴趣，借由此书正好可以了解一下别人是如何理解以及如何实践 Web 开发的，更何况作者还是豆瓣的工程师，因此对书的内容还是有一些期待。

经过一番阅读之后，我与伟明交流了一些看法，他给予了详细的解释与说明，我对他的写作思路也有了一些了解。对 Web 开发的理解其实可以有很多角度，比如，从开发者的角度，这就会更多从具体的功能实现、框架使用来看待；从运维者的角度，会更多地从部署、维护、平台的角度来理解；从测试及质量的角度，会关心代码的测试性及代码审查；从框架开发者的角度，就要了解 Web 开发涉及哪些领域，每一领域应该用什么技术与工具来组织，不同领域又如何通过某些框架来有机地结合在一起。仅凭一本书，想完全满足所有人的需求是非常困难的。

阅读本书，我最大的感受就是：全和新。

全指的是内容覆盖面较广。原本我以为作者会主要讲 Flask 框架的开发，但其实 Flask 框架在本书中的比重并不大，反而是与 Web 相关的开发技术的介绍占了大部分的篇幅，甚至也包含了部署以及 Python 本身的一些特性和工具。对此我也有疑问，并向作者咨询。Web 开发的概念其实太大了，不同的角度可以有不同的理解。比如我们常说的 Web 框架，其实绝大部分都只涉及展示相关的开发，所以应该更精确地称之为 Web 展示框架或 Web 应用框

架。但是它很有可能依赖底层的批处理、大数据处理等技术，这些虽然不能算纯粹的 Web 技术，但是却可以放在 Web 开发这一概念下。因此如果把每一块与 Web 开发相关的内容都写出来，那么本书的厚度就可想而知了。所以作者是从个人实践的角度出发，把他所理解的与 Web 开发相关的技术尽可能全地，并且尽可能用更多的实例来讲述。之所以我会有“全”这个感受，因为书的内容涉及了 Web 框架、Ajax 的前后端交互、测试、数据库、数据分析、服务化、部署、系统管理、常用工具等内容，有点百科全书的意味。

为什么说“新”，因为书中讲的许多东西都是现在正在流行的技术或工具，像 Flask、Celery、Jupyter、Supervisor、SaltStack、Pandas 等。其中有些我还是第一次接触，说明作者平时接触的内容的确非常丰富，同时也结合了豆瓣的一些具体的实例，这样会更有借鉴意义。

全书的难度不是很大，内容广泛全面，不过因为篇幅所限，对于前端的技术介绍得不多，有些章节可能描述也不是太细。不过前端技术虽然也算是 Web 开发技术，但是与 Python 的关系就不那么紧密了，本书毕竟是一本 Python 相关的书，所以涉及不多也是正常的。而且许多具体的技术本身内容都很丰富，也绝不是短篇幅可以说清楚的，所以反而有个基础性的介绍，在需要时自行学习可能更好。因此本书比较适合对于 Web 开发有一定了解，但是希望了解更多 Python Web 开发技术的读者。

非常感谢作者把自己的经验分享给大家。

李迎辉

Python 开源资深行者

Python-CN 邮件列表创建人

UliPad 和 Uliweb 作者

业界热评

本书从 Python 开发开始，循序渐进，把网站工程的全貌展现在读者的眼前，是了解 Web 工程从开发到上线完整流程的绝佳参考书籍。同时书中的很多实例取自豆瓣工程开发团队的实际工作，对于想了解豆瓣内部技术实现的朋友，也有很大的参考价值。

邢犇（CNBorn）
前豆瓣东西技术负责人

开卷有益，已经很久没有看到原创的有价值的 Python Web 开发书籍了。很多刚进入 Python 世界的人，想要在 Web 开发上有更多的发展，但却不知如何往下学习。伟明的这本书提供了一个非常好的“知识地图”，书中涉及了 Python Web 开发的方方面面。与此同时，对于那些已经在 Web 开发上积累了一些经验，想要更进一步学习的人来说，这本书也能让你收获满满。我阅读完书稿也有了收获。书中涉及的知识点非常多，任何一个点都可以单独写成一本书。作者根据自己的经验积累，提炼出干货，略去了基础的部分，这对于读者来说也是幸事，不然你可能得抱一个大部头的书回去了。最后需要说的是，在 Web 开发的道路上，这本书是不错的进阶指南。

胡阳（the5fire）
Python 程序员
目前就职于手机搜狐网
任资深开发工程师
负责 m.sohu.com 网站的前后端开发和维护

董伟明是我见过的实践和执行能力超群的工程师。这本书从开发环境的搭建，Web 框架的使用，到最后的持续集成和 Python 的进阶用法，无一不是他多年的实际工程经验总结，十分宝贵。如果你刚开始学习 Python，这本书能给你展示 Python 的方方面面，让你可以快速

进入实际的 Web 工程的开发。如果你已经使用 Python 多年，这本书也能让你学习到 Python 的很多使用技巧。

姚钢强 (acmerfight)

知乎工程师

这本书非常全面地介绍了使用 Python 进行 Web 开发的方方面面，既有 Web 框架、缓存、消息队列、并发处理的场景介绍和技术选型，又有开发流程、质量保证的丰富实战经验。作者通过非常细致的 Step by Step 教程，一步一步揭开了 Web 开发的神秘面纱，不管你有没有 Web 开发基础，相信都能从这本书中获益良多。

蔡斌 (VeryCB)

DeepDevelop 工程师

前豆瓣条目组技术负责人

本书适合有一定 Python 和 Web 开发基础的用户。书中没有对语言基础的讲解，更多的是对 Web 方面的专注。内容很丰富，基本上覆盖日常 Web 项目开发中涉及各个层级，对相关概念和原理的描述十分详尽，而每个示例代码都进行了分段解释，清晰明了。

正如书名，整本书都是作者对实际 Web 项目中大量实战经验的总结，绝非纸上谈兵。相信通过阅读该书可以帮助开发者规避掉大量项目中的“坑”，构建出更高性能、更稳定的 Web 项目。

强烈推荐从事 Web 开发的 Pythoner 阅读。

Spawnris

腾讯工程师

前言

为什么写这本书

2011 年，我还在一家互联网商务公司做一枚小小的运维工程师，那时公司的运维使用的语言主要是 Shell。我本来是一个网络运维，后来在工作中开始接触 Shell，学了 2 个月之后，感觉可以应付各种需求，虽然程序运行得很慢，但作为一个工作不久且不是 IT 相关专业毕业的运维人员，我还是有点沾沾自喜。这种情况只持续了 2 个多月，由于公司高速发展，一个新入职的同事打破了我的美梦。

这位新同事入职的第一件工作就是对接各个业务部门的日志需求。很快就发生了一件让我特别震撼的事情：同样的一个日志需求，我使尽浑身解数用 20 行 Shell 写好，运行一次要 20 分钟，而这位同事使用 Perl 语言的脚本只用了 4 行，运行 3 分钟就完成了。可以想象我当时的感受。这是我第一次了解到选择正确的工具和方法的重要性。我抑制不住地告诉当时的领导悦秋：我要学一门运维使用的高级语言！

悦秋特别郑重地告诉我：一定要学 Python。而在 2011 年，Python 还只是一门小众语言，在 BAT 等大公司招聘时仅作为一些职位的附属要求。回想至此，非常感谢悦秋让我选择了正确的路，否则我现在可能只能写关于 Perl 语言实践方面的书了。

从运维到运维开发，再到豆瓣做产品开发（也就是 Web 开发），一路走来我发现，自己走了很多弯路，没有人告诉我什么是对的，什么是错的，该怎么做选择。这些都得自己花时间去琢磨和验证，有时候从 Google、StackOverflow 搜索答案，或者在 GitHub 直接看源码获得灵感甚至正确答案，而涉及职业规划、该学什么、怎么学，这样的问题除了悟性大部分就是靠直觉了。

从买书看基本语法和拷贝别人的代码开始学习 Python，很早我就开始努力让代码符合 PEP8，尽量让代码写得 Pythonic（这点很关键，未来就不再需要费力改正学习过程中留下的坏习惯了）。能用 Python 完成日常工作之后，我开始研究和寻找各种 Python 高级玩法、黑魔法。这个时候我还是在不断买书、看书、看之前买的书、看一些技术博客来巩固和补充自己的知识体系。所谓“技术”中的“术”也就到这里了。

有了“术”还远远不够，还需要有实际的经验，以及在正确的时机使用正确的工具和方法，这是“技”。“技”是一套分析并解决问题的思路，要想提高“技”，除了个人的领悟，最重要的是靠大量的实践，有时候我们称之为“造轮子”。关键是在造的过程中得思考，比如什么时候该抽象了？这个轮子和竞品相比有什么优势？技术选型上为什么要使用 XX？

使用 Python 会遇到这样的问题：什么时候该用多进程？怎样提高代码执行效率？Flask 为什么流行？曾经遇到一个冷门的 Celery 的 Bug，当时使用谷歌没有找到解决方案，甚至解决思路也没有，怎么办？我开始翻阅 PEP 文档，阅读优秀开源项目的源码，还把 Python 标准库模块中的代码全部过了一遍，收获颇丰。同时我还会根据工作中遇到的问题，给开源项目和 Python 提一些 Issue，后来还给它们提交 Patch，用自己微薄的能力，让社区变得好一点点。

虽然过了而立之年，我还在不间断地更新博客（dongwm.com），希望给其他开发者带来帮助和灵感。当许艳编辑找到我时，双方一聊，发现对国内开发者而言实战类 Python Web 开发方面的书确实不多，我顿时觉得可以以自己多年的工作经验积累写一本，为女儿两岁生日送上一份不一样的礼物。作为一个做过运维，现在做后端，却经常写前端程序的人来说，我了解产品的整个过程，是适合写一本这样的书的。写这本书的意义还在于，将自己这几年来使用 Python 进行 Web 开发中对各方面知识的理解和积累的经验进行梳理和总结，让更多人受益，同时对自己也是一种成长，也算是对国内的 Python 环境做出个人的贡献了。

谁应该看本书

虽然语言只是工具，但是阅读本书还是需要有一定的 Python 基础，如果你还没学过 Python，那就先学习一段时间再来阅读本书，收获会更多。

本书主要面向如下 4 类人群：

- Web 开发者。
- 使用 Python 语言的运维工程师和运维开发工程师。

- 想提高 Python 技能的开发者。
- 想了解 Python Web 开发的其他开发者。

为什么值得看

本人阅读过大量和 Python 有关的纸质书和开源图书，渐渐学到了很多控制自己“剁手”买书的方法。我来分析一下为什么你值得拥有本书。:)

为什么要买书来看？我认为不外乎两个原因：有趣和能学到东西。技术书肯定不会太有趣，那么最重要的就是能学到东西。市面上 Python 相关的书相当多，但是有些内容陈旧或者不符合国情，或者并非开发第一线的人所写或者翻译，这样的书显然价值就要打一些折扣；其次是同质化严重，偏入门级别，我个人认为市面上关于 Python 入门或者教授语法知识的书不少，而再深入一点的就很匮乏了。

本书有几个特点：第一，使用了当前主流和前瞻性的技术，如 Docker、Ubuntu 16.04 LTS、Cython、CFFI、Py.test、asyncio、IPython 5.0 LTS 等，书中一部分内容是在 Python 3 下完成的。本书中全部工具都使用当前最新版，能保证在相当长的时间内书中的内容都不会过时；第二，笔者在国内应用 Python 最大的豆瓣网做产品开发，一直在第一线写代码，大量例子和经验都来自实际工作；第三，笔者非常关注 GitHub 和 Python 社区，会第一时间了解到新的趋势和思想，并在书中体现。举个例子，代码检查工具 pep8 已经在 Guido van Rossum 的要求下改名为 pycodestyle 了。

叔本华在《人生的智慧》中说过一段话，大意是“人要么庸俗，要么孤独”。笔者认为这个道理在阅读上面也成立：读什么样的书，就会逐渐成为什么样的人。本书提供了很多笔者在其他书中没有看到过的思考方式和 Python 的用法，这也是本书存在的意义。

本书的特别用法

本书中有些效果可能会让读者迷惑，在这里解释一下。

1. 交互模式下的效果。本书在交互模式下完成的例子都使用了 IPython，效果如下：

```
In : template.render(name='Xiao Ming')
Out: u'Hello Xiao Ming!'
```

其中 In : 是输入，Out: 是输出。

2. 终端提示符。本书绝大多数的终端提示符没有使用 # 或者 \$，而使用了 >。

本书的组织方式和阅读建议

笔者和身边的一些朋友交流过，大多数人买书来看，基本上都是看到书中讲到了自己一直不太懂的知识点，或者感兴趣的话题。因此，在写作本书时，笔者有意让每章相对独立。你可以选择跳着看，当然更推荐从第 1 章一直看到最后一章，因为本书是按照一个 Web 产品从无到有、从简单变复杂、从基础到进阶的过程编排的。

我们先来大致了解一下这个过程。

- **第 1 章** 首先回答两个问题：“为什么应该选择 Python 作为 Web 开发语言”和“选择 Python 2 还是 Python 3”，然后介绍 Python 主流的 Web 框架，并为如何选择给出建议。
- **第 2 章** 帮助读者跑起来一个包含本书所讲内容的 Ubuntu 环境，读者可以直接在里面运行书中的例子。限于篇幅，如果想要了解环境搭建的整个过程以及笔者做这些选择的理由，可以在本书源代码项目中的 `setup.md` 文件中获取，短链接地址为 <http://bit.ly/29N4Pqv>。接着将展开介绍 Python 的包管理和虚拟环境相关的内容。通过学习这章，读者对 Python 生态环境会有一定了解。
- **第 3 章** 先从最简单的 Flask 例子开始，学习一些 Flask 相关的知识，接着学习 Jinja2 和 Mako 模板（Mako 在豆瓣的使用非常广泛），使用 MySQL，最后学以致用，从零开始完成一个相对复杂、在豆瓣有类似功能的文件托管服务。这个项目贯穿本书，在之后的章节中会对它继续扩展。
- **第 4 章** 这一章是 Flask 的进阶，包含了大量的 Flask 扩展的使用，还介绍了信号机制和 Werkzeug 的使用。到这里读者对 Flask 和 Web 开发已经入门，可以根据自己的想法自己做一些应用了。
- **第 5 章** 现在 Web 端应用对交互的要求很高，移动应用对后端的 API 需求也非常多，需要很好的异构通信方式，本章将介绍笔者对 REST 的理解，并提出一些设计 API 的注意事项，最后通过 jQuery 和 fetch 实现使用 Ajax 的例子，让读者了解如何让前后端通信。
- **第 6 章** 我们已经有了有实际业务逻辑的 Web 应用，可是用户还不能访问，本章将介绍如何选择应用服务器，用主流的方式在生产环境中运行这个应用。之前应用中只是使用了 MySQL，在实际的网站应用中，缓存、键值对数据库、NoSQL 数据库都是主流的解决方案，本章将一一介绍为什么要用这种技术以及怎么用。最后作为总结，笔者根据自己的实际经验绘制一张大型网站的架构图，并详细介绍其中模式选

择的理由和经验。

- **第7章** 在第6章，Web应用已经运行起来，用户也可以访问了。但是如下问题也随之出现：

- 应用依赖多个服务，如MySQL、Redis等，这些服务器在新环境中的部署是有顺序要求的，而且程序要保证一直在运行状态。
- 上线过程不能自动化。每次上线都要手动执行很多命令，既耗时又容易出错。
- 希望能及时了解和分析服务器和应用的运行状态。

看完本章相信你就可以知道对应的解决方案了。

- **第8章** Web应用运行良好，可是应用的质量还没有保证，如何在上线之前发现更多的Bug的需求变得越来越迫切。本章将介绍主流的测试方法，并用一个GitHub项目实现持续集成。
- **第9章** 前面介绍的是Web应用必备的内容，从本章开始介绍一些进阶的内容。消息队列能带来更好的用户体验，本章将介绍豆瓣用到的消息队列工具Beanstalkd，以及Celery推荐的消息队列RabbitMQ。如果Web产品有大量的定时任务或者其他异步任务，就可以使用Python界最知名的Celery解决，本书将从浅入深让读者熟悉Celery原理和使用方法，最后分享笔者使用的进阶实践。
- **第10章** 现在各个大公司都在谈服务化，通过这几年的改造和实践，大家都有自己的一套服务化方案，豆瓣也不例外。本章将告诉读者为什么要服务化、豆瓣的服务化设计，以及使用开源的Thrift改造文件托管服务。
- **第11章** 笔者在工作中经常要给各个业务方提供数据支持，如日志统计分析、数据报表。本章将演示如何使用纯Python代码在单个服务器上利用多核实现MapReduce功能，还详细讲解豆瓣工程师都在用的DPark，包含安装、环境配置、使用和框架化分析UV & PV；接着将展示几个笔者在实际工作中遇到过的数据报表需求，并讲解如何用Pandas做数据可视化。
- **第12章** 这一章将详细介绍IPython和Jupyter Notebook这两个工具，并分享其在豆瓣对应的实践。除此之外，还列出笔者日常用来排错和调试的工具，包括了解Linux服务器的相关情况、性能测试、分析Python程序性能瓶颈三个方面。
- **第13章** Web开发日常也会有一些并发编程工作，所以本章以抓取微信公众号文章为主线，分别使用多线程、多进程、Gevent、Future和asyncio这5种编程方式完成不同阶段的爬取任务，也深入地分析在它们之间如何选择。
- **第14章** Python进阶并不只针对Web开发人员，对于所有Python开发者都有意义。

本章介绍了一些非常有用的标准库模块，也有笔者对《Python 之禅》的理解和总结的一些语法实践，还讲述了从 Python 3 移植一些有用的功能及编写 Python 扩展等内容。

- **第 15 章** 介绍笔者日常进行 Web 开发的流程和经验，还着重介绍了多个代码质量保证工具，以及豆瓣的一些质量保证实践。最后一节，笔者将谈谈代码评审的意义和实际经验。

需要说明的是，章与章以及每章内的节与节之间没有明确的递进关系，不同产品形态让 Web 产品在其发展的不同阶段对技术的选择和侧重点都有所不同。举个例子，并不是前 8 章讲到的内容在产品中都用到了，才能在实际应用中引入消息队列和 Celery，要根据实际情况灵活选择。在产品发展还没有遇到瓶颈之前就要考虑和尝试引入对应的解决方案，确保不会影响产品高速发展。

使用代码示例

书中的完整例子的代码都存放在 GitHub 上（https://github.com/dongweiming/web_develop），可以在 GitHub 上查看和下载。本书提供的 Vagrant 和 Docker 环境已经包含了这个项目，但可能并不是最新的代码，如果遇到运行失败等问题，可以在项目的 Issue 页面先搜索是否之前有人遇到过同样问题。如果没有找到，请尝试保存本地修改后使用如下命令同步最新的代码，然后再运行：

```
git pull --rebase origin master
```

例子文件的名字在书的对应位置会提到，文件存放在对应的章节的子目录下，比如第 3 章第 1 节的例子，就会放在 `chapter3/section1` 目录下。有一些文件是相对于项目根目录的，比如静态文件统一存放在 `static` 目录下，模板文件统一存放在 `templates` 目录下。绝大多数模板文件使用相同的对应章节的约定，比如第 4 章第 2 节的模板存放在 `templates/chapter4/section2` 目录下。

反馈和勘误

由于笔者水平有限，加之本书的编写稍显仓促，书中难免出现一些不准确甚至错误的理解和不能运行于所有环境的例子，恳请读者批评指正。如果你在阅读过程中有任何问题或者发现任何错误，都可以去本书源代码的 Issue 页面来提问，或者发送邮件到 ciici123@gmail.com，我会尽量一一解答。

本书的全部勘误将放在项目的 `errata.md` 文件中，地址是 <http://bit.ly/2a9ep8V>，在提交错误之前请先看看是不是之前有人已经提过了。

为了促进本书读者的交流和反馈，笔者创建了一个关于 Python 和 Web 开发等主题的社区，地址是 <http://python-cn.org>，关于本书的意见和建议也可以到这里反馈。这个平台也能帮助你认识更多阅读本书、对 Python 和 Web 开发有兴趣和有经验的人。

致谢

首先感谢给本书做技术评审的洪强宁（hongqn）、大妈（Zoom.Quiet）、清风、李迎辉（limodou）、邢犇（CNBorn）、胡阳（the5fire）、姚钢强（acmerfight）、蔡斌（VeryCB）和 Spawnris。你们从技术和全书规划等角度提出了很多有用的意见和建议，让我受益匪浅。尤其特别感谢 Zoom.Quiet，提供了大量建设性的批评意见，这些意见深刻而透彻，对笔者改进和重构内容帮助极大，尤其是对第 3 章和第 5 章。接下来，感谢写书过程中豆瓣同事对我的帮助：感谢 Guillaume Bouriez 对 PIDL 部分的审阅，他以法国人的严谨提出了很多有用的建议；感谢潘妙才让豆瓣的缓存服务运行良好，产出了非常好的使用缓存的文档，并对我提出的缓存问题耐心回答；感谢田忠博对我搭建 DPark 提供了很大的帮助，对 DPark 特点的总结源于田老师的内部分享幻灯片。

感谢电子工业出版社的编辑许艳和其他同事，本书能在 8 个月的时间出版，离不开你们的敬业精神和一丝不苟的工作态度。

最后，最应该感谢的是我的父母、妻子和女儿，你们是我生命中最重要的人。尤其感谢我的妻子，在我占用大量周末、晚上的时间进行写作的时候，能够给予极大的宽容、支持和理解，对我和女儿悉心照顾并承担了全部的家务，让我能够全身心投入到写作中。

Amy，你是我写作本书最大的动力，这是爸爸送给你 2 周岁的特殊礼物。

2016 年 8 月 7 日

目录

第 1 章 初识 Python Web 开发	1
Python Web 开发介绍	1
为什么应该选择 Python 作为 Web 开发语言	2
选择 Python 2 还是 Python 3	2
Web 框架介绍	3
主流 Web 框架	3
小众的 Web 框架	5
选择 Web 框架时应遵循的原则	5
第 2 章 Web 开发前的准备	7
搭建一个能运行的虚拟机环境	7
安装 VirtualBox	8
使用 Vagrant 安装	8
使用 Docker 安装	10
包管理和虚拟环境	13
包管理	13
使用 pip 替代 easy_install	13
distribute、distutils 和 setuptools	14
entry_points	15
插件系统	16

虚拟环境	17
virtualenv	18
virtualenv 定制化	18
virtualenvwrapper	21
virtualenv-burrito	23
autoenv	24
进阶篇：pip 高级用法	25
命令自动补全	25
普通用户安装	25
编辑模式	25
使用 devapi 作为缓存代理服务器	26
PYPI 的完全镜像	27
第 3 章 Flask Web 开发	28
Flask 入门	29
安装 Flask	29
从 Hello World 开始	29
配置管理	31
调试模式	32
动态 URL 规则	32
自定义 URL 转换器	33
HTTP 方法	34
唯一 URL	35
构造 URL	36
跳转和重定向	36
响应	38
静态文件管理	40
即插视图	40
蓝图	43
子域名	43
命令行接口	44

模板	46
Jinja2	46
Mako	52
使用 MySQL	60
安装 MySQL 和驱动	61
设置应用账号和权限	61
用 MySQLdb 写原生语句	62
事务提交和回滚	63
ORM 简介	64
使用 SQLAlchemy	65
使用 ORM	67
数据库关联	69
在 Flask 中使用 SQLAlchemy	71
记录慢查询	73
理解 Context	74
本地线程	74
Werkzeug 的 Local	75
flask.request	76
使用上下文	77
使用 LocalProxy 替代 g	80
从零开始实现一个文件托管服务	80
首页	84
重新设置图片页	86
下载页	87
预览页	87
短链接页	88
 第 4 章 Flask 开发进阶	 89
Flask 的信号机制	89
Blinker 的使用	89
Flask 中内置的信号	90

自定义信号	92
信号订阅的高级用法	92
Flask-Login 中的信号	93
Flask 的扩展	95
Flask-Script	95
Flask-DebugToolbar	97
Flask-Migrate	98
Flask-WTF	100
Flask-Security	102
Flask-RESTful	109
Flask-Admin	111
Flask-Assets	115
Werkzeug 的使用	118
DebuggedApplication	118
数据结构	120
功能函数	121
密码加密	122
中间件	123
 第 5 章 REST 和 Ajax	 127
什么是 REST	127
RESTful API 设计指南	128
使用名词来表示资源	128
关注请求头	129
合理使用请求方法和状态码	129
正确地使用 REST	130
对输出的结果不再包装	131
不要做出错误的提示	131
使用嵌套对象序列化	131
版本	132
URI 失效和迁移	132

信息过滤	132
速度限制	133
缓存	133
并发控制	134
使用 Ajax	135
第 6 章 网站架构.....	140
Python 应用服务器	140
WSGI 协议	141
常见的 WSGI 容器	141
Web 服务器 Nginx	143
Web 服务器与应用服务器的区别	143
为什么要选择 Nginx	143
安装 Nginx	144
使用 Nginx 部署 Flask 应用	144
缓存系统 Memcached	149
Libmc 安装配置	150
使用原生 SQL 缓存	152
缓存更新策略	157
Memcached 使用的经验	157
键值对数据库 Redis	157
操作 Redis	158
Redis 应用场景	159
分片和集群管理	168
NoSQL 数据库 MongoDB	169
为什么使用 NoSQL	169
MongoDB	169
使用 pymongo 的例子	171
使用 Mongoengine 的例子	174
MongoDB 实践经验	176

大型网站架构经验	182
缓存	183
负载均衡	183
高可用	184
业务拆分	184
集群	184
第 7 章 系统管理.....	186
进程管理 Supervisor	186
Supervisor 组件	187
配置 Supervisor	187
使用 Supervisor	190
应用部署 Fabric	193
Fabric 应用接口	194
使用 Fabric 管理 Flask 应用	197
配置管理工具 SaltStack 和 Ansible	199
SaltStack	200
Ansible	207
使用 Psutil	213
使用 Sentry 收集错误信息	215
安装配置 Sentry	216
启动 Sentry	218
创建团队和项目	218
配置 SDK	220
使用 StatsD、Graphite 等搭建 Web 监控	223
配置 Graphite	225
使用 StatsD	226
配置 Diamond	227
发布指标项	227
使用 Grafana	228
使用 Kenshin	232

第 8 章 测试和持续集成	233
使用 unittest 和 doctest 做测试	233
unittest	233
doctest	236
使用 py.test 和 mock	237
py.test	237
mock	241
持续集成	243
使用 Tox 集成	248
 第 9 章 消息队列和 Celery.....	250
使用 Beanstalkd	251
使用 Beanstalkc	252
深入理解 RabbitMQ	253
AMQP	254
虚拟主机	258
插件系统	258
通过 Web 和 REST API 管理 RabbitMQ	259
故障转移	262
使用 Celery	262
Celery 的架构	263
Celery 序列化	265
安装配置 Celery	265
从一个简单的例子开始	265
指定队列	268
使用任务调度	269
任务绑定、记录日志和重试	270
在 Flask 应用中使用 Celery	271
深入理解 Celery	274
Celery 的依赖	274
任务调用	277

信号系统	278
Worker 管理	279
监控和管理 Celery	280
子任务	281
进阶篇：Celery 最佳实践	283
使用自动扩展	283
善用远程 Debug	283
合理安排任务周期	284
合理使用队列和优先级	285
保证业务逻辑的事务性	285
关闭你不想要的功能	285
使用阅后即焚模式	285
善用 Prefetch 模式	286
善用工作流	286
第 10 章 服务化.....	288
为什么需要服务化	288
RPC 框架	289
服务化带来的问题	290
微服务架构	290
使用 Thrift	291
定义 IDL 文件	292
服务端实现	294
客户端实现	297
PIDL——豆瓣的服务化实践	301
PIDL 架构	302
第 11 章 数据处理.....	305
使用 MapReduce 做日志分析	305
使用 MapReduce	305

使用 DPark	309
分布式文件系统 MooseFS	309
Mesos	310
配置 DPark 环境	311
从 WordCount 开始	314
PV & UV 统计	316
数据报表	320
发送带有样式和附件的邮件	320
创建 xlsx 文件	325
使用 Pandas	328
Pandas 入门	329
读取 MySQL 数据库	332
和 Flask 应用集成	332
 第 12 章 帮助工具.....	 336
IPython	336
IPython 交互模式	338
常用的 Magic 函数	338
配置和自定义 IPython	341
IPython 的扩展系统	342
使用 IPython 调试复杂代码	343
双进程模型	344
并行计算	345
Jupyter Notebook	347
Notebook 格式	350
Notebook 格式转换和预览	351
为什么使用 RequireJS	352
在 Notebook 里使用 Echarts	353
富显示	355
自定义 JavaScript 和 CSS 样式	356
使用 nbextension 扩展 Notebook	358

在 Notebook 上使用并行计算	359
调试和 Debug 工具	360
了解 Linux 服务器运行情况	360
性能测试	366
Python 程序性能分析	369
性能调优实践	373
进阶篇：定制基于 IPython 的交互解释环境	374
进阶篇：豆瓣东西的 Jupyter Notebook 实践	376
 第 13 章 Python 并发编程	 383
使用多线程	385
使用 Gevent	392
使用多进程	399
使用 Future	406
使用 asyncio	408
async/await	409
Future	412
使用 aiohttp	414
使用队列	416
 第 14 章 Python 进阶	 418
使用标准库模块	418
errno	419
subprocess	420
contextlib	421
glob	424
operator	424
functools	426
collections	428
Python 语法最佳实践	432
命名	434

使用 join 连接字符串	435
EAFP vs LBYL	435
定义类的 __str__/_repr__ 方法	436
优美的 Python	437
从 Python 3 移植	439
partialmethod	439
singledispatch	440
suppress	442
redirect_stdout/redirect_stderr	443
使用 CFFI/Cython 编写 Python 扩展	444
使用 CFFI	444
使用 Cython	447
进阶篇：使用 PyObjC 发送通知	451
 第 15 章 Web 开发项目实践	455
Web 项目经验总结	455
开发流程	455
使用合理的项目结构	456
关注代码复杂度	457
代码质量保证工具	457
Pycodestyle 对中文缩进的处理	458
Flake8	459
Pylint	460
其他代码质量保证工具	461
使用 AST 做静态检查	461
其他静态检查工具	467
编写 Flake8 扩展	468
代码评审的意义	470
作为被评审者	471
作为评审者	472
评审的标准	473

第 1 章

初识 Python Web 开发

本章将回答 Python 工程师关心的如下 3 个问题：

- 为什么应该选择 Python 作为 Web 开发语言？
- 在 Python 2 和 Python 3 之间如何选择？
- 在这么多的 Python Web 框架中哪些是主流的，它们的特点是什么，该如何选择？

Python Web 开发介绍

打开本书，说明你对 Python Web 开发是有兴趣的，或者正打算开始学习使用 Python 做 Web 开发，又或者已经是一个 Web 开发者。那么，用 Python 做 Web 开发的人员需要具备哪些技术能力呢？

笔者列了一个清单：

- 至少熟悉一种 Python Web 框架。
- 熟悉 Python 语法。
- 熟悉数据库、缓存、消息队列等技术的使用场景、使用方法等。
- 日常能使用 Linux 或 Mac 系统工作。
- 有性能调优经验，能快速定位问题。
- 对 HTML/CSS/JavaScript 有一定了解，有使用经验。

Web 开发需要掌握的知识很广，而且对每个知识点都要有深入的了解。除此之外，还要对业务有深刻理解，并能写出可维护性足够高的代码。

为什么应该选择 Python 作为 Web 开发语言

对于 Web 开发，有很多的编程语言可以选择，为什么应该选择 Python 呢？

在 2016 年 7 月的 TIOBE 编程语言排行榜（<http://bit.ly/2a5jikR>）中，Python 已经升至第 4 位，可见 Python 现在有多么流行。现在无论 PC 端还是移动互联网的 Web 开发工作，对产品做的各种尝试都需要更快地拿出模型并进行迭代，创业公司尤甚。Python 语言更好地符合了时代的需求，所以它也受到了越来越多的关注，越来越多的人接受 Python，并在生产环境中使用它。个人认为 Python 非常适合做 Web 开发，理由如下：

1. Python 是一门优雅而健壮的编程语言，它继承了传统编译语言的强大性和通用性，同时借鉴了简单脚本和解释型语言的易用性。Python 非常适合做快速的原型开发，很多场景下的性能问题可以通过使用 C/C++ 写 Python 扩展等方式优化解决。
2. Python 应用广泛，在大数据、算法、运维等领域都有不错的对应工具和库，可以有效降低产品流程中不同职位之间的技术壁垒，团队人员的沟通更容易，解决问题也更快。
3. Python 标准库和第三方的库很强大，有非常多的知名项目都是用 Python 编写的。
4. 从 2005 年 Django 开源，2008 年 Reddit 开源，到 2010 年 Flask 开源，Python 用作 Web 开发已经有着 10 多年的历史，国内的豆瓣、搜狐，国外的 Reddit、YouTube、Instagram、Pinterest、Bitbucket、Disqus、Dropbox 等公司都选择 Python 作为 Web 开发的语言（<http://bit.ly/28QKXBv>）。不用担心 Python 可靠性与性能问题，因为它已经经受了时间和大规模用户并发访问的考验。

选择 Python 2 还是 Python 3

首先需要强调的是，编程其实重在对编程思想的理解和经验的积累，Python 2/3 的思想基本是共通的，只有少量的语法有差别甚至不兼容。当对 Python 熟悉到一定程度时，即使只会 Python 2 也可以在很短的时间就能掌握 Python 3 代码的编写。

Python 社区曾经于 2014 年初在 python-dev（<http://bit.ly/28RIJDV>）、hacker news（<http://bit.ly/28RU1YZ>）等渠道针对这个问题做过一个调查，部分结果如下（<http://bit.ly/28QKyUe>）：

1. 97.51% 的用户还在写 Python 2 的代码。
2. 60% 的用户在写 Python 3 的代码。

3. 78.09% 的用户更多地写 Python 2 的代码。
4. 77.09% 的用户认可 Python 3。

有以下主流操作系统已默认使用 Python 3:

- Arch Linux (<http://bit.ly/28RlNDN>)
- Ubuntu 16.04 LTS (<http://bit.ly/28QL8Nm>)
- Fedora (<http://bit.ly/28RUdHy>)

Django 2.0 将不再支持 Python 2。

现在 Python 2 只是在做一些 Bug 修复、新硬件和操作系统兼容性相关的维护工作，不再有新的功能加入。但是话说回来，虽然 Python 2 只会支持到 2020 年 (<http://bit.ly/28QL4ND>)，但是对于生产环境，尤其是重要的应用，不会为了这么一个理由就迁移到 Python 3，只能是在新写的项目中抛开这个历史包袱。所以在未来相当长的时间内，Python 2 都会存在。但如果你愿意拥抱变化，义无反顾地选择 Python 3 吧。如果是为了满足现在工作中的需要，尤其是依赖的软件只能运行在 Python 2 下，那还是首选 Python 2。

Web 框架介绍

Python 的 Web 框架可算是百花齐放，各种框架和微框架数不胜数，关于 Python 框架孰优孰劣的讨论一直没有间断过。这种争论给 Web 开发工程师们带来了很大的困扰，尤其对初中级的工程师来说，不知道如何选择。本节我们就来介绍当前知名的 Web 框架的特点及其应用场景。

主流 Web 框架

主流的 Web 框架有以下几种。

Django

Django 最初是被开发用来管理劳伦斯出版集团旗下一些以新闻内容为主的网站的，它以比利时的吉普赛爵士吉他手 Django Reinhardt 来命名，它和 Flask 是使用最广泛的 Python Web 框架。Django 能如此知名很大程度上是因为提供了非常齐备的官方文档，它提供了一站式的解决方案，包含缓存、ORM、管理后台、验证、表单处理等，使得开发复杂的数据库驱动的网站变得很简单。但正因为它坚持自己对于 Web 框架的理解，系统耦合度太高，替换掉内置的功能往往需要花费一些功夫，所以学习曲线也相当陡峭。

Flask

Flask 是一个轻量级 Web 应用框架，它基于 Werkzeug 实现的 WSGI 和 Jinja2 模板引擎。Flask 的作者是 Armin Ronacher，本来这只是作者愚人节开的一个玩笑，但是开源之后却受到 Python 程序员的喜爱，目前在 GitHub 上的 Star 数量已经超过了 Django。它的设计哲学和 Django 不同：只保留核心，通过扩展机制来增加其他功能。Flask 用到的依赖都是 Pocoo 团队开发的。Pocoo 团队其他的项目还有 Pygments、Sphinx、以及 lodgeit。Flask 的扩展环境非常繁荣，基本上 Web 应用的每个环节都有对应的扩展供选择，就算没有对应的扩展也能很方便地自己实现一个。

Pyramid

Pyramid 在国内知名度并不高，主要原因是中文文档匮乏，其高级用法需要通过阅读源代码获取灵感。尽管被 Django 和 Flask 的光芒遮蔽，但是它的性能要比 Flask 高。Pyramid 的灵感来源于 Zope、Pylons 1.0 和 Django。在我们的传统观点里，小框架通常牺牲了大框架的特色，反之亦然。但是事实上我们不能真正把控一个应用程序最终的发展情况，而 Pyramid 在努力朝着胜任不同级别应用的框架的方向在走。虽然它默认使用 Chameleon 和 Mako 模板，但很容易切换到 Jinja2，甚至可以让多种模板引擎共存，通过文件后缀名来识别。豆瓣赞赏和豆瓣钱包等产品就是基于此框架实现的，代码量级和 Flask 相同，性能比 Flask 要略高。

Bottle

Bottle 也是一个轻量级的 Web 框架。它的特点是单文件，代码只使用了 Python 标准库，而不需要额外依赖其他第三方库。它更符合微框架的定义，截止到今天只有 4100 多行的代码。

Tornado

Tornado 全称 Tornado Web Server，最初是由 FriendFeed 开发的非阻塞式 Web 服务器，现在我们看到的是被 Facebook 收购后开源出来的版本。它和其他主流框架有个明显的区别：它是非阻塞式服务器，而且速度相当快。得益于其非阻塞的方式和对 epoll 的运用，Tornado 每秒可以处理数以千计的连接，这意味着对于长轮询、WebSocket 等实时 Web 服务来说，Tornado 是一个理想的 Web 框架。

Web.py

Web.py 也是一个微框架，由 Reddit 联合创始人、RSS 规格合作创造者、著名计算机黑客 Aaron Swartz 开发。Web.py 使用基于类的视图，简单易学却功能强大。

小众的 Web 框架

Quixote

Quixote 是由美国全国研究创新联合会（Corporation for National Research Initiatives, CNRI）的工程师 A.M.Kuchling、Neil Schemenauer 和 Greg Ward 开发的一个轻量级 Web 框架，它简单、高效、代码简洁。豆瓣的大部分用户产品都使用定制版的 Quixote 作为 Web 框架，虽说这有历史原因（当时 Django/Flask 等框架还没有出现），但是也证明这个始于 2000 年的框架是可以经受时间考验的。它使用目录式的 URL 分发规则，用 Python 来写模板。PTL 模板更适合程序员，但是并不适合美工参与前端代码的编写和修改，豆瓣在开发中使用了 Mako 替代 PTL。不建议在生产环境选用 Quixote。

Klein

Klein 是 Twisted 组织开源出来的基于 `werkzeug` 和 `twisted.web` 的微框架。Flask 很不错，但是不能使用异步非阻塞的方式编程，根本原因是它和 Django、Pyramid 一样，都基于 WSGI，而 WSGI 的接口是同步阻塞的。Klein 用法非常像 Flask，却可以使用异步的方式开发 Web 应用。

选择 Web 框架时应遵循的原则

介绍了这么多的框架，那么在工作中怎么选择呢？以下是笔者总结的一些原则：

1. 选择更主流的框架。因为它们的文档更齐全，技术积累要更多，社区更繁盛，能得到更好的支持。
2. 关注框架的活跃情况。关注项目的更新频率、Issue 和 Pull Request（在本书中都简称 PR）的响应情况。如果一个项目已经很长时间没有更新了，或者有一堆的问题需要解决但是没有得到回应，就不应该将这样的框架放在生产环境中。
3. 确认选择的框架是否足够满足需求。没有最好的框架，只有最合适的框架。你所选择的 Web 框架不仅需要满足当前的需求，也要充分考虑项目发展一段时间之后的情况，即前瞻性。如果在做选择时有个人喜好这样的因素，需要确认自己有能力对选择的 Web 框架提供支持，避免盲目选择而导致将来推倒重来的情况。
4. 注意媒体时效性。在做选择的时候可能会参考网络上的一些文章，但是需要注意其发表时间。举个例子，看了一篇 2012 年的博客，里面说应该选择 A 而不是 B，并给了多个理由。而现在的情形可能已经发生了变化：B 经过很久的努力已经做得更优秀或者 2012 年之后出现了更优秀的 C。

5. 客观看待媒体的观点。媒体的观点并不一定是正确的（或者不是全部正确），如果不是官方的说明，就应该保持怀疑和谨慎的态度，取其精华去其糟粕，切勿完全拿来主义，应该真正做实践验证之后再决定。

对于上述的 Web 框架，可以参考如下几条建议：

1. 如果是为了供演示或者不太考虑长期发展，只是需要用到数据库的 CURD 操作、写 REST API 之类的简单型应用，框架的选择非常宽松，用着顺手的即可。
2. 如果你初学 Web 框架，建议选择 Flask 作为入门框架，学习曲线相对 Django 而言要低很多。等熟悉 Flask 之后再学习 Django，就会容易很多。
3. Pyramid 和 Django 都是面向大型应用的，Pyramid 更灵活，开发者空间大得多，值得考虑。
4. 网站性能出现问题时，往往问题不只出在 Web 框架和编程语言上，做选择时在环境中按实际产品逻辑测试一段时间即可得到结论，但是不要只相信看到的一些性能对比文章，尤其不要做无意义的 Hello World 级别的测试。

第2章

Web 开发前的准备

本章介绍 Web 开发的一些准备工作，主要包含如下内容：

- 环境的准备，以便读者能够使用 Vagrant 或者 Docker 提供的 Ubuntu 环境运行书中的例子。
- 介绍包管理工具 pip 及一些高级用法。
- 实现 PYPI 的缓存代理和完全镜像。
- 使用 virtualenv 及其扩展实现虚拟环境管理。

搭建一个能运行的虚拟机环境

Ubuntu 是 Linux 发行版里面被用作个人桌面最多的系统，现在已经有很多公司选择使用 Ubuntu Server 作为生产环境的操作系统。

笔者选择了发布于 2016 年 4 月 21 日，版本代码名为“Xenial Xerus”的 Ubuntu 16.04 LTS。LTS 是 Long Term Support（长期支持）的缩写，这样的版本一般桌面版官方支持 3 年，服务器版支持 5 年。



下面使用 Docker 和 Vagrant 安装 Ubuntu 的章节是根据写作本书时的官方文档编写的，当你阅读到这里的时候有可能某些内容已经变更，请以官方文档为准。

安装 VirtualBox

VirtualBox 是 Oracle 开源的虚拟化系统，它支持 Linux、OS X、Windows 等平台，Docker 和 Vagrant 环境都需要使用它作为宿主机。到官方网站下载对应平台的最新版并安装。安装过程很傻瓜化，按提示一步一步执行到安装完成即可。

使用 Vagrant 安装

为什么选择 Vagrant？原因如下：

1. Vagrant 是一个操作虚拟机的工具，它会很快地完成一套开发环境的部署，也解决了各个开发环境不一致的问题，减少了重复配置环境而造成的时间和精力上的浪费。举个例子，在没有用 Vagrant 之前，新员工加入后常常需要一到两天的时间搭建完整的开发环境，而有了 Vagrant，直接启动就好了。先不需要了解所有相关环境的知识和细节，在工作中再慢慢熟悉就行了。
2. 它底层支持 VirtualBox、VMware 甚至 AWS 作为虚拟机系统，可以满足不同用户的需要。
3. 可以通过“vagrant provision”，使用 Shell 脚本或者主流的配置管理工具（如 Puppet、Ansible 等）对软件进行自动安装、更新和配置管理。

安装 Vagrant

目前 Rubygems 上 Vagrant 只更新到 1.5.0，不支持 VirtualBox 5.0 及以上版本。需要到官方网站选择对应的平台下载并安装。

安装完成后检查一下是否安装成功：

```
> vagrant --version
Vagrant 1.8.3
```



需要使用 Vagrant 1.8.3 及以上版本支持 Ubuntu 16.04 LTS。

使用 Vagrant

一个打包好的操作系统在 Vagrant 中称为 Box，实际上它是一个 zip 包，包含了 Vagrant 的配置信息和 VirtualBox 的虚拟机镜像文件。

默认的 Ubuntu 系统需要进行配置，如改成使用 aliyun 的源，安装 Python 等软件，添加用户 ubuntu 等。为了方便读者，可以使用笔者打包好的 Box（dongweiming/web_develop）。首先，克隆本书源代码并进入项目目录：

```
> git clone https://github.com/dongweiming/web_develop
> cd web_develop
```

项目中包含了 Vagrantfile 文件，不需要初始化：

```
Vagrant.configure(2) do |config|
  config.vm.box = "dongweiming/web_develop"
  config.vm.hostname = "WEB"
  config.vm.network :forwarded_port, guest: 9000, host: 9000
  config.vm.network :forwarded_port, guest: 3141, host: 3141
  config.vm.network :forwarded_port, guest: 5000, host: 5000
  config.ssh.username = "ubuntu"
  config.ssh.password = "ubuntu"
  config.ssh.insert_key = false
  config.ssh.private_key_path = ["~/.ssh/id_rsa"]
  config.vm.provision "file", source: "~/.ssh/id_rsa.pub", destination: "~/.ssh/
    authorized_keys"
  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--name", "web_dev", "--memory", "1536"]
  end
  config.vm.network "public_network", bridge: "en0: Wi-Fi (AirPort)"
end
```

Vagrantfile 保存了虚拟机的各项配置。上述设置的用途如下：

1. 使用笔者定制的 Box，名字为 dongweiming/web_develop。
2. 设置虚拟机的主机名。
3. forwarded_port 用来设置端口转发，“guest: 9000, host: 9000”表示访问本机 9000 端口的流量就会转发到虚拟机上的 9000 端口，反之亦然。
4. 没有使用 vagrant 作为默认用户，这是因为代码中有些指定了目录地址，为了兼容 Docker 容器，使用中立的用户 ubuntu。
5. customize 语句把虚拟机在 VirtualBox 中的显示名改为 web_dev，内存为 1536 MB。
6. config.vm.network 用来添加一个桥接网卡，它将使用 DHCP 获得 IP。

定制的 Box 基于官方的 ubuntu/xenial64，笔者在实际使用中发现不能直接登录，这里我们将创建一个 SSH 密钥用于自动登录。如果之前没有创建过 SSH 密钥，需要执行如下命令并回车：

```
> ssh-keygen
```

执行完毕会生成 `~/.ssh/id_rsa`（密钥）和 `~/.ssh/id_rsa.pub`（公钥）。

现在启动虚拟机：

```
> vagrant up
```

启动的时候会检查本地是否有这个 Box，没有的话就会下载。所以第一次花的时间会比较长。

第一次启动完成后需要使用配置脚本来初始化系统环境：

```
> vagrant provision
==> default: Running provisioner: file...
```

`provision` 会执行 Vagrantfile 中定义的 `file` 命令，把本机的 `~/.ssh/id_rsa.pub` 拷贝到目标服务器并保存为 `~/.ssh/authorized_keys`。

启动完成就可以登录虚拟机了：

```
> vagrant ssh
```

登录之后就可以直接验证本书提到的内容了。

使用 Docker 安装

Docker 是 dotCloud 开源的一个使用 Go 语言编写的基于 Linux 容器（Linux Containers, LXC）的容器引擎。它有以下优点：

- 性能卓越。在以前的虚拟化方案中，虚拟机都是一个完整的操作系统，这本身就会占用 CPU、内存、硬盘等资源。而 Docker 是“操作系统级别的虚拟化”，可以达到秒级启动。IBM 曾发表过一篇关于虚拟机和 Linux Container 性能对比的论文（[http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/\\$File/rc25482.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf)），论文中实际测试了虚拟机和 LXC 在 CPU、内存、网络的负载等方面的情况，结果显示 Docker 容器本身几乎不占用什么开销。
- 可移植性带来了工作效率的提升。没有 Docker 之前，运维和开发工作中经常会发生这样的情况：本地开发环境正常，但是上线就出问题。这就需要花时间找原因然后解决。而 Docker 是“一次封装，到处运行”，开发者只需要关注开发，运维人员只需要关注部署，比如要做服务器迁移，重新部署和调试这样的工作就可以省去了——在新的服务器上启动需要的容器就可以了。

安装 Docker

Docker 官网的文档非常详细。打开菜单第一栏 “Install”，根据你当前使用的桌面系统选择对应的版本：如果使用 Max OS X，就选择 “Installation on Mac OS X”；如果使用 Ubuntu，就先打开 “Linux” 子菜单，再选择 “Installation on Ubuntu”。也可以选择直接安装 Docker 的二进制文件 “Installation from binaries”。

这时候可能会显示对你选择的系统的要求。比如选择 “Ubuntu”，会提示支持如下版本的 Ubuntu：

```
Ubuntu Xenial 16.04 (LTS)
Ubuntu Wily 15.10
Ubuntu xenial 14.04 (LTS)
Ubuntu Precise 12.04 (LTS)
```

如果你使用的系统不在上述列表中，不代表不能正常使用 Docker，只是如果出现问题可能不会得到官方的修复支持，所以笔者建议升级到它要求的版本范围。

对于 Max OS X，按照官方文档进行，然后点击 “Docker Quickstart Terminal” 选项就可以进入 Docker Shell。它会进行一系列的初始化，最后会提示：

```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

上面提到的 192.168.99.100 是 Docker 创建的虚拟机的 IP，之后访问应用其实都是在请求这个 IP。如果忘记了也可以通过如下命令获得：

```
> docker-machine ip
192.168.99.100
```

如果看到类似如下的输出就说明 Docker 安装成功了：

```
$ docker --version
Docker version 1.11.2, build b9f10c9
```

下载镜像

在 Docker 中镜像称为 Image。为了节省读者的时间，笔者已经上传了基于 Ubuntu:16.04 LTS 的镜像。

1. dongweiming/web_develop:dev: dev 标签包含了基本环境，比如使用 aliyun 的源，安装 Python、Git、IPython、net-tools 等软件，克隆本书源码，添加用户 ubuntu 并默认使用 ubuntu 这个用户等。所做的全部前期工作都可以参考项目下的 Dockerfile 文件。这个版本适合从零开始跟着笔者一起完成每一节例子的读者。

2. `dongweiming/web_develop`: 也就是默认的 `latest` 标签, 它可以直接运行绝大多数应用和例子。这个版本会经常更新, 适合有一定基础, 想直接看到运行效果或者会跳跃着看书的读者。

执行如下命令会自动下载镜像:

```
> docker pull dongweiming/web_develop:dev
```

Docker 会查看镜像是否已经加载到 Docker 主机上, 如果没有, 它就会从镜像仓库 Docker Hub 下载这个镜像。下载完成后可以看到类似如下的镜像列表:

```
> docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
dongweiming/web_develop dev          43fb02d9c1a3     2 weeks ago     292.2 MB
dongweiming/web_develop latest       5895e8d64924     34 seconds ago  6.075 GB
```

要进入容器应该使用如下命令:

```
> docker run --name web_dev -it -p 9000:9000 -p 3141:3141 -p 5000:5000 dongweiming/
web_develop /bin/zsh
```

这个命令有如下含义:

- `--name` 指定了容器的名字为 `web_dev`, 如果不指定, 将由系统随机取一个名字。
- `-p` 可以显式地暴露特定端口, 比如 `9000:9000`, 就表示 `web_dev` 这个容器里面的 `9000` 端口可以通过 `192.168.99.100:9000` 访问。
- `/bin/zsh` 是登录容器的默认 Shell。

进入容器后, 默认使用 `ubuntu` 这个用户, 并切换到 `/home/ubuntu/web_develop` 目录下。

从容器退出之后, 容器就关闭了, 重新登录容器的方法如下:

```
> docker start web_dev # 启动容器
web_dev
> docker attach web_dev # 要回车2次
```

Docker 虚拟机端口转发

之前提到, 在 Docker 环境中访问应用需要使用 `http://192.168.99.100:PORT` 这个地址, 由于 Vagrant 做了端口转发, 直接在宿主主机上访问 `http://127.0.0.1:PORT` 即可。那么, 可不可以统一呢? 当然可以。

首先获得 Docker 虚拟机的名字:

```
> docker-machine inspect|grep MachineName
"MachineName": "default",
```

Docker 虚拟机的名字是 default。通过如下的 Shell 命令即可添加本机与容器的端口镜像：

```
> for port in 3141 5000 9000
do
VBoxManage controlvm "default" natpf1 "tcp-port$port,tcp,127.0.0.1,$port,, $port"; echo
    $port
done
```

VBoxManage 是 Virtualbox 提供的命令行管理工具，通过它能完成很多用 GUI 不能实现的工作。现在已经可以用 `http://127.0.0.1:PORT` 这样的地址访问了。

包管理和虚拟环境

包管理

利用 Python 工作不可避免需要使用第三方包，目前安装第三方包的方法有以下三种：

- 通过 Python 社区开发的 pip、easy_install 等工具。
- 使用系统本身自带的包管理器（yum、emerge、apt-get 等）。
- 通过源码安装（python setup.py install）。

其中的第一种方法，最推荐使用 pip。

第三方包主要分布在 The Python Package Index（<https://pypi.python.org/pypi>）官方的仓库（简称 PYPI）、GitHub、Bitbucket 等代码托管服务上。

使用 pip 替代 easy_install

pip 是一个用来安装和管理 Python 包的工具，它是 easy_install 的替代品，也是目前社区的主流工具。之所以能成为主流，有以下的原因：

- pip 已经内置到 Python 2.7.9 和 Python 3.4 及其以上的版本里面。
- easy_install 只支持安装，没有提供卸载、展示当前已安装的包列表等功能。
- pip 支持二进制包使用 wheel 格式（后缀是.whl），而 easy_install 不支持。
- pip 能非常好地支持虚拟环境工具 virtualenv。
- 支持多种版本工具格式的包的下载和安装。
- 可以集中管理项目依赖列表（文件名字一般叫作 requirements.txt），使用 -r 选项安装这些依赖。

我们使用的虚拟机默认没有安装 pip，可以使用如下方式安装它：

```
> sudo apt-get install python-pip -yq
```

系统自带的 pip 版本比较低，可使用 pip 的自更新来升级：

```
> sudo pip install pip -U -q # -q表示静默安装，减少过程输出
> pip --version
pip 8.1.2 from /usr/local/lib/python2.7/dist-packages (python 2.7)
```



笔者建议平时也经常这样将 pip 更新到最新版。

distribute、distutils 和 setuptools

在 Python 发展史上出现了很多创建和发布包的工具。当你想要把自己的项目分享出去，放到 PYPI 或者其他托管服务上的时候，就需要借助这样的工具来构建和分发项目。我们先了解一下常见的 3 个工具。

- **distribute**: setuptools 的一个分支，在 setuptools 0.7 时被合并回 setuptools，现在已经被弃用。
- **distutils**: 早在 1998 年它就被内置到 Python 标准库里，但是只提供了有限的支持。
- **setuptools**: 它是用来解决 distutils 的限制的替代品，但是需要额外的安装。和 distutils 相比，它有很多优点。
 - 可以创建 Eggs 和 Wheel (<https://wheel.readthedocs.org/en/latest/>) 格式的包。
 - 自带 easy_install，能帮助你找到、下载、安装以及更新需要使用的包。
 - 支持 PYPI 上传，可以很方便地把本地项目发布到 PYPI。
 - 支持测试集成。
 - 提供了更多的功能函数和额外特性。

除非项目的环境依赖简单到只需要用到 distutils 就可以了，否则推荐使用 setuptools 包。如果是一个开源项目，建议使用类似下面这样的兼容代码：

```
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup
```


还有几个值得一提的工具：

- distlib (<https://bitbucket.org/pypa/distlib>)，它正致力于取代 distutils，未来有希望进入标准库。应该对 distlib 保持关注。
- pbr (<https://launchpad.net/pbr>)，全称“Python Build Reasonableness”，是 setuptools 的辅助工具，最初是为 OpenStack 开发的，它借鉴了 distutils 2，利用 setup.cfg 文件承载元数据，下面提到的 virtualenvwrapper (<http://bit.ly/28UjAMb>) 就使用了 pbr 打包。

entry_points

发布的包经常需要一个（或多个）可执行的入口，以使用户直接执行和调用。比如 flake8，安装之后在终端就可以执行 flake8 这个命令：

```
> cat `which flake8`  
#!/usr/bin/python  
  
# -*- coding: utf-8 -*-  
import re  
import sys  
  
from flake8.main import main  
  
if __name__ == '__main__':  
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])  
    sys.exit(main())
```

回忆一下，我们安装的时候并没有特别地复制一个文件到/usr/local/bin/flake8，这是怎么实现的呢？要感谢 setuptools 提供了解决方案。先看一下 setup.py (<http://bit.ly/28RVQVG>) 中的 entry_points 部分：

```
entry_points={  
    'distutils.commands': ['flake8 = flake8.main:Flake8Command'],  
    'console_scripts': ['flake8 = flake8.main:main'],  
    'flake8.extension': [  
        'F = flake8._pyflakes:FlakesChecker',  
    ],  
}
```

其中有个 console_scripts 的键，表示注册一个叫作 flake8 的系统命令，这个命令会调用 flake8.main 的 main 函数，安装的时候由 setuptools 来帮助我们生成了/usr/local/bin/flake8 这个文件。选择这种方式，而不是直接复制文件，是基于如下原因：

- 没办法预先知道 Python 解释器的版本和位置。
- 很难确定会安装在哪里。
- 无法优雅地解决可移植到不同系统上的问题。

当然你可能看到过这样的写法：

```
#!/usr/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'flake8==2.5.1','console_scripts','flake8'
__requires__ = 'flake8==2.5.1'
import sys

from pkg_resources import load_entry_point

if __name__ == '__main__':
    sys.exit(
        load_entry_point('flake8==2.5.1', 'console_scripts', 'flake8')()
    )
```

它和上面命令的作用是一样的，只不过它是使用 `easy_install` 或者 “`python setup.py install`” 安装生成的，`load_entry_point` 可以去抽取入口点是 “flake8” 的键并定位到 `flake8.main`，然后运行 `main` 函数。

使用 `entry_points` 的优点，就是可以让这些入口点能够被其他 Python 程序动态发现包所提供的功能，但是对应的代码的耦合度非常低。

插件系统

`entry_points` 机制还能直接作为插件系统，比如 `flake8` 项目，上面提到它的 `entry_points` 中有这样一段：

```
'flake8.extension': [
    'F = flake8._pyflakes:FlakesChecker',
]
```

`flake8` 的文档中介绍了如何写一个插件（<http://flake8.readthedocs.org/en/latest/extensions.html>），就是使用了 `entry_points` 作为插件机制。我们看一下已有的 `flake8` 插件之一 `pep8-naming`（<https://github.com/PyCQA/pep8-naming>）的 `entry_points` 的一部分（<https://github.com/PyCQA/pep8-naming/blob/master/setup.py#L47>）：

```
'flake8.extension': [
    'N8 = pep8ext_naming:NamingChecker',
]
```

我们先安装 pep-naming:

```
> pip install --user pep8-naming
> flake8 --version
2.5.1 (pep8: 1.5.7, pyflakes: 0.8.1, naming: 0.3.3, mccabe: 0.3.1) CPython 2.7.6 on
Linux # 可以看到已经出现了naming: 0.3.3
```

它们使用了相同的键 `flake8.extension`, 是怎么工作的呢? 看一下 `flake8` 的插件实现 (篇幅所限, 此处只截取了相关的代码):

```
import pep8
from pkg_resources import iter_entry_points

extensions = []

def _load_entry_point(entry_point, verify_requirements):
    if hasattr(entry_point, 'resolve') and hasattr(entry_point, 'require'):
        if verify_requirements:
            entry_point.require()
        plugin = entry_point.resolve()
    else:
        plugin = entry_point.load(require=verify_requirements)

    return plugin

for entry in iter_entry_points('flake8.extension'):
    checker = _load_entry_point(entry, verify_requirements=False)
    pep8.register_check(checker, codes=[entry.name])
    extensions.add((checker.name, checker.version))
```

这样就实现了一个简单的插件系统。



要查看当前环境全部的入口点, 可以使用 `iter_entry_points(None)`。

虚拟环境

Python 工程师工作中常常遇到这样的场景: 系统自带的 Python 是 2.6, 却需要用到 Python 2.7 中的某些特性; 不同的项目之间使用不同版本的某些包, 但是因为某些原因 (比如有依赖冲突) 却不能都升级到最新版本。

所有的包都共用一个目录，很容易出现不小心更新了项目 A 的依赖，却影响了项目 B 用到的依赖的情况。这个时候就需要对环境进行隔离，使用虚拟环境让全局的 `site-packages` 目录非常干净和可管理。

Python 社区中创建和管理虚拟环境的工具有 `virtualenv` 和 `pyenv`。这些工具可以帮助你快速创建一个单独、干净的 Python 环境，你可以把所需的包安装到各自孤立的环境中。

virtualenv

先安装 `virtualenv`：

```
sudo pip install virtualenv
```

现在创建一个 Python 环境：

```
> virtualenv venv
New python executable in /home/ubuntu/venv/bin/python
Installing setuptools, pip, wheel...done.
```

`virtualenv` 默认会创建一个包含了 Python 可执行文件、常用的标准库、激活 `virtualenv` 环境的脚本的目录。

使用 `source` 激活 `virtualenv` 环境：

```
> source venv/bin/activate
```

```
(venv)> which python # 注意终端提示的改变，前面添加了“(venv)”前缀。
/home/ubuntu/venv/bin/python
```

可以看到已经不再使用系统环境变量中的 Python 了。如果要退出虚拟环境，可以取消激活：

```
(venv)> deactivate
```

virtualenv 定制化

`virtualenv` 默认只是生成一个非常标准的 Python 环境，而在实际使用中，项目都会有第三方包的依赖，会出现多个项目依赖相同的包的情况。举个例子，当项目有严格的代码规范要求 and 集成测试要求时，那么每个项目可能都需要 `py.test`、`flake8` 之类的基础包，通常还会添加 `IPython` 这样的增强交互工具。除此之外，项目还可能有初始化的需要，要求在创建虚拟环境的时候做一些额外的工作。那么，对于这种情况，就可以生成一个定制的 `virtualenv` 脚本。

本节我们展示一个在生成 `venv` 环境的同时安装 `flake8` 的自定义脚本。首先让 `ubuntu` 这个用户对 `virtualenv` 文件可写，方便我们直接替换：

```
> sudo chown ubuntu:ubuntu `which virtualenv`
```

生成自定义脚本的内容如下 (create-venv-script.py):

```
import subprocess

import virtualenv

virtualenv_path = subprocess.check_output(['which', 'virtualenv']).strip()

EXTRA_TEXT = '''
def after_install(options, home_dir):
    subprocess.call(['{}/bin/pip'.format(home_dir), 'install', 'flake8'])
'''

def main():
    text = virtualenv.create_bootstrap_script(EXTRA_TEXT, python_version='2.7')
    print 'Updating %s' % virtualenv_path
    with open(virtualenv_path, 'w') as f:
        f.write(text)

if __name__ == '__main__':
    main()
```

生成定制的 virtualenv 脚本:

```
> python chapter2/section2/create-venv-script.py
Updating /usr/local/bin/virtualenv
```

/usr/local/bin/virtualenv 已经被替换成定制的版本了。如果不想全局替换, 可以把 virtualenv_path 换成其他路径。

现在生成一个虚拟环境, 就会自动安装 flake8 了:

```
> virtualenv tmp
New python executable in /home/ubuntu/web_develop/tmp/bin/python2.7
Also creating executable in /home/ubuntu/web_develop/tmp/bin/python
Installing setuptools, pip, wheel...done.
...
Installing collected packages: mccabe, pyflakes, pep8, flake8
Successfully installed flake8-2.5.4 mccabe-0.4.0 pep8-1.7.0 pyflakes-1.0.0

> ll tmp/bin/flake8
-rwxrwxr-x 1 ubuntu ubuntu 239 May 26 09:51 tmp/bin/flake8
```

Virtualenv 定制脚本接受 3 个扩展函数。

- `extend_parser(optparse_parser)`: 添加额外的选项。
- `adjust_options(options, args)`: 改变当前的选项。
- `after_install`: 在默认的环境安装好之后, 执行其他工作, 主要通过这个函数完成定制。

现在做一个更复杂的定制, 实现如下需求:

- 增加一个选项, 可以额外安装指定的包。
- 假如没有设置这个选项, 在终端发出警告提示 (warn)。
- 默认的虚拟环境都安装在指定的目录之下 (~/venv)。

看一下升级之后 `EXTRA_TEXT` 的内容 (`create-venv-script_v2.py`):

```
EXTRA_TEXT = '''
ROOT_PATH = '/home/ubuntu/venv'

def extend_parser(parser):
    parser.add_option(
        '-r', '--req', action='append', type='string', dest='reqs',
        help="specify additional required packages", default=[])

def adjust_options(options, args):
    if not args:
        return

    base_dir = args[0]
    args[0] = join(ROOT_PATH, base_dir)

def after_install(options, home_dir):
    if not options.reqs:
        logger.warn('Warn: You maybe need specify some required packages!')

    for req in options.reqs:
        subprocess.call(['{}/bin/pip'.format(home_dir), 'install', req])
...'''
```



`adjust_options` 中能直接用 `join` 是因为 `virtualenv` 初始化的时候已经设置了 `join = os.path.join`。

重新生成 virtualenv:

```
> python ~/web_develop/chapter2/create-venv-script2.py
Updating /usr/local/bin/virtualenv
```

看一下升级后的效果:

```
> virtualenv -h|grep req # 增加的新选项
-r REQS, --req=REQS    specify additional required packages

> virtualenv tmp
New python executable in /home/ubuntu/venv/tmp/bin/python2.7
Also creating executable in /home/ubuntu/venv/tmp/bin/python
Installing setuptools, pip, wheel...done.
Warn: You maybe need specify some required packages!
```

把 tmp 安装到了 /home/ubuntu/venv 目录下, 最后发出警告提示。

现在创建一个自动安装 Flake8 和 Jinja2 的虚拟环境:

```
> virtualenv tmp2 -r flake8 -r jinja2
```

virtualenvwrapper

virtualenvwrapper 是对 virtualenv 的功能扩展, 它有如下用途:

- 用来管理全部的虚拟环境。
- 能方便地创建、删除和拷贝虚拟环境。
- 用单个命令就可以切换不同的虚拟环境。
- 可以使用 Tab 补全虚拟环境。
- 支持用户粒度的钩子支持。

使用如下方式安装:

```
> sudo pip install virtualenvwrapper
```

先初始化 virtualenvwrapper:

```
> export WORKON_HOME=~/.venv
> source /usr/local/bin/virtualenvwrapper.sh
```

初始化之后 ~/.venv 目录也会添加一些用户级别的 virtualenvwrapper 的钩子模板。通常上述两行会放在 shell 的配置里面, 比如 ~/.zshrc, 这样每次登录时就自动初始化了。

初始化时给系统添加了一些虚拟环境相关的函数，比如用来创建虚拟环境的 `mkvirtualenv`：

```
> mkvirtualenv venv1
(venv1)> deactivate # 退出虚拟环境的命令还是一样的
```

`mkvirtualenv` 还会添加如下 5 个项目级别的钩子模板。

- `predeactivate`：在虚拟环境取消激活之前执行。
- `postdeactivate`：在虚拟环境取消激活之后执行。
- `preactivate`：在虚拟环境激活之前执行。
- `postactivate`：在虚拟环境激活之后执行。
- `get_env_details`：使用 `lsvirtualenv/showvirtualenv` 等命令时，对于当前环境的额外钩子，可以添加虚拟环境介绍等内容。

再创建一个新的虚拟环境，并添加一个钩子，在取消激活之后输出 “Deactivated!”：

```
> mkvirtualenv venv2
(venv2)> cat /home/ubuntu/venv/venv2/bin/postdeactivate
#!/usr/bin/zsh
# This hook is sourced after this virtualenv is deactivated.
echo 'Deactivated!'

(venv2)> deactivate
Deactivated!
```

现在已经有两个虚拟环境了，可以使用 `Tab` 管理它们：

```
> workon <Tab> # 输入workon， 然后按Tab
venv1      venv2
> workon venv1 # 选择venv1
(venv1)> workon venv2 # 直接使用workon就可以切换到venv2
(venv2)>
```

其他常用的命令

- `lsvirtualenv`：列出全部的虚拟环境。
- `showvirtualenv`：列出单个虚拟环境的信息。
- `rmvirtualenv`：删除一个虚拟环境。
- `cpvirtualenv`：拷贝虚拟环境。
- `allvirtualenv`：对当前虚拟环境执行统一的命令。比如，要给 `venv1` 和 `venv2` 都安装 `flake8`，就可以用 `allvirtualenv pip install flake8`。

- `cdvirtualenv`: 可以直接切换到虚拟环境的子目录里面。

```
(venv1)> cdvirtualenv bin
```

```
(venv1)> pwd  
/home/ubuntu/venv/venv1/bin
```

- `cdsitepackages`: 和 `cdvirtualenv` 同理, 切换到虚拟环境的 `site-packages` 目录下。
- `lssitepackages`: 列出 `site-packages` 目录下的目录。

用户级别的钩子脚本

用户级别的钩子脚本都在 `VIRTUALENVWRAPPER_HOOK_DIR` 这个变量的目录下, 默认是 `WORKON_HOME` 变量指定的目录。

常用的钩子脚本如下。

- `get_env_details`: 当不带参数执行 `workon` 时执行。
- `postmkvirtualenv`: 在虚拟环境创建和激活之后执行。
- `preactivate`: 在虚拟环境激活之前执行。
- `postactivate`: 在虚拟环境激活之后执行。
- `predeactivate`: 在虚拟环境取消激活之前执行。
- `postdeactivate`: 在虚拟环境取消激活之后执行。
- `prermvirtualenv`: 在虚拟环境删除之前执行。
- `postrmvirtualenv`: 在虚拟环境删除之后执行。

更多的钩子脚本类型可以在这里找到: Per-User Customization (<http://bit.ly/28ReOu6>)。

virtualenv-burrito

`virtualenv-burrito` (<http://bit.ly/28UjSml>) 是一个安装、配置 `virtualenv` 和 `virtualenvwrapper` 及其依赖的傻瓜式工具。无须上面烦琐的过程, 只需要一步即可:

```
> curl -sL https://raw.githubusercontent.com/brainsik/virtualenv-burrito/master/  
virtualenv-burrito.sh | $SHELL
```

它自动把初始化脚本放在 `/home/ubuntu/.zprofile` 里面, 这样在下次登录之后就可以使用了。如果想立刻使用初始化脚本, 可以用如下语句:

```
source /home/ubuntu/.venvburrito/startup.sh
```

startup.sh 会自动创建了 ~/.virtualenvs 作为 WORKON_HOME，用法和 virtualenvwrapper 是一样的。

使用如下命令即可对 virtualenv 和 virtualenvwrapper 进行更新：

```
> virtualenv-burrito upgrade
```



本书所有 Python 2 的例子都使用了这个虚拟环境。

autoenv

autoenv (<http://bit.ly/290YIQG>) 让你在切换目录的时候可以完成自动激活虚拟环境（如果有的话）等定制操作。

先安装它：

```
> sudo pip install autoenv
> source /usr/local/bin/activate.sh
```

我们需要做的就是再目录下新建一个 .env 文件，然后修改这个文件，添加激活命令就好了：

```
> touch .env
> echo "source source /home/ubuntu/.virtualenvs/venv/bin/activate" > .env
```



这里一定要用完整的绝对路径，否则执行 cd web_develop/subdir 之类的命令会让虚拟环境激活失败。

效果如下：

```
> cd # 先切换到其他目录
> cd web_develop # 出现如下提示
autoenv:
autoenv: WARNING:
autoenv: This is the first time you are about to source /home/ubuntu/web_develop/.env:
autoenv:
autoenv:     --- (begin contents) -----
autoenv:     source /home/ubuntu/venv/bin/activate
autoenv:
autoenv:     --- (end contents) -----
autoenv:
autoenv: Are you sure you want to allow this? (y/N) y
(venv) > # 可以看到提示符的变化
```

进阶篇：pip 高级用法

命令自动补全

pip 支持自动补全功能，对于 zsh 用户非常友好。zsh 用户使用如下命令就可以支持自动补全了：

```
> pip completion --zsh >> ~/.zprofile
> source ~/.zprofile
```

或者在 ~/.zshrc 里面加一行：

```
eval "`pip completion --zsh`"
```

如果使用 bash，则应该执行如下命令：

```
pip completion --bash >> ~/.profile
```

现在输入“pip i<Tab>”就会变成“pip install”，自动补全了 install 这个子命令。

普通用户安装

如果不是 root 用户，没有 sudo 权限，也不在虚拟环境里面，要如何安装包呢？可以使用命令：

```
> pip install django --user
```

这样，安装的包会放在当前用户的.local 目录下：

```
> pip show django|grep Location
Location: /home/ubuntu/.local/lib/python2.7/site-packages
```



不能在 virtualenv 中使用“--user”，需要退出虚拟环境后才可以使⽤。

编辑模式

作为开发者，为了让最新的开发版代码及时生效，开发过程中可以使用 pip 的编辑模式：

```
> git clone https://github.com/dongweiming/vine
> cd vine
> pip install -e .
```

```
> cd
> python -c 'import vine; print vine'
<module 'vine' from '/home/ubuntu/web_develop/vine/vine/__init__.py'>
```

可以看到，“pip install -e.”会把开发目录作为包的路径。这样就可以实时修改了。

使用 devapi 作为缓存代理服务器

pip 缓存只针对当前的用户。如果公司使用 Python 的规模很大，尤其是有很多自己分发的包的时候，使用缓存代理是非常提高下载效率的方法，这样就不再依赖网络环境到 PYPI 下载包了。

```
> pip install devpi-server
```

启动 devpi-server:

```
> devpi-server --host=0.0.0.0 --start
...
starting background devpi-server at http://0.0.0.0:3141
...
```

默认缓存服务器使用了 3141 端口。现在我们就可以使用 -i 参数指定使用缓存代理了:

```
> pip install -i http://localhost:3141/root/pypi/ tornado
```

当然，可以把这个 “index-url” 设置写到配置文件中:

```
> mkdir ~/.config/pip -p
> cat ~/.config/pip/pip.conf
[global]
index-url = http://localhost:3141/root/pypi/+simple/
```

安装 Django 看一下:

```
> pip install django
Collecting django
  Downloading http://localhost:3141/root/pypi/+f/706/87b2d4a2dfa51/Django-1.9.6-py2.py3
    -none-any.whl (6.6MB)
    100% |████████████████████████████████████████| 6.6MB 42.1MB/s
Installing collected packages: django
Successfully installed django-1.9.6
```

可以看到现在的安装包是从缓存代理获取的，而不是从每次都到 PYPI 下载了。

如果希望像 PYPI 那样有个 Web 界面，可以安装 devpi-web:

```
> pip install -U devpi-web
```

重启 devpi-server:

```
> devpi-server --host=0.0.0.0 --stop
> devpi-server --host=0.0.0.0 --start
```

现在就可以通过 <http://127.0.0.1:3141/> 访问这个简单的 Web 界面了。

PYPI 的完全镜像

bandersnatch (<https://bitbucket.org/pypa/bandersnatch>) 是 PyPA 组织根据 PEP 381 (<http://www.python.org/dev/peps/pep-0381/>) 实现的镜像客户端。它可以帮我们建立一个包含了全部包的本地镜像服务。

先安装 bandersnatch:

```
(bandersnatch)> pip install -r https://bitbucket.org/pypa/bandersnatch/raw/stable/
requirements.txt
(bandersnatch)> pip install zc.buildout
(bandersnatch)> sudo apt-get install mercurial -yq
(bandersnatch)> hg clone https://bitbucket.org/pypa/bandersnatch ~/bandersnatch
(bandersnatch)> ~/bandersnatch
(bandersnatch)> buildout
```

默认还没有创建 bandersnatch.conf 这个配置文件, 假如不使用 “-c”, 默认是 /etc/bandersnatch.conf。先启动 bandersnatch 创建配置文件:

```
(bandersnatch)> bandersnatch -c bandersnatch.conf mirror
```

更新 directory 配置项, 修改 directory 选项的值为 /data/pypi, 这个目录就是存放包的根目录:

```
(bandersnatch)> grep directory e |grep -v ';'
directory = /data/pypi
```

再次启动 “bandersnatch mirror”:

```
(bandersnatch)> sudo mkdir /data
(bandersnatch)> sudo chown ubuntu:ubuntu /data
(bandersnatch)> bandersnatch -c bandersnatch.conf mirror
```

现在就开始同步镜像了。需要注意, PYPI 上所有的包加起来有几百 GB, 而且还在不断增加中, 请合理安排硬盘资源。

第 3 章

Flask Web 开发

Flask 是非常流行的 Python Web 框架，它能如此流行，原因主要有如下几点：

- 有非常齐全的官方文档，上手非常方便。
- 有非常好的扩展机制和第三方扩展环境，工作中常见的软件都会有对应的扩展。自己动手实现扩展也很容易。
- 社区活跃度非常高。
- 微框架的形式给开发者更大的选择空间。
- Poccoo 团队出品，Flask 和相关依赖（Jinja2、Werkzeug）的设计很优秀。比如使用装饰器配置路由、用 Blueprint 实现模块化、请求/应用上下文等。

Flask 主要依赖三个库。

- Jinja2：默认的模板引擎。
- Werkzeug：一个包含 WSGI、路由、调试的工具集。
- Itsdangerous：基于 Django 签名模块（<http://bit.ly/28QV7Fb>）的签名实现。

Flask 本身尽量保持了内核的精简，其设计初衷就是不会替开发者做太多决策，而且就算 Flask 已经帮你做出选择也能很容易地替换。举两个例子：

- Web 程序不可避免要和数据库打交道，使用 SQLAlchemy、MongoEngine、不用 ORM（对象关系映射）而直接基于 MySQL-python 这样的底层驱动进行开发都是可以的，选择权完全在你的手中。
- 把默认的 Jinja2 模板引擎替换成 Mako 或者其他模板引擎都非常容易。

本章包含如下内容：

- 通过多个应用例子了解 Flask 框架使用的一些精髓。
- 介绍目前最流行的模板引擎 Jinja2 和 Mako 的使用,以及实践总结和做选择时的建议。
- 通过一些例子让读者熟悉 MySQLdb 的使用,并演示如何与 Flask 应用集成。
- 通过源码帮助读者理解 Flask 的上下文设计,并演示在大型应用中使用上下文钩子的例子。
- 通过一个真实的案例学以致用。先分析需求,接着从零开始实现一个文件托管应用。

Flask 入门

安装 Flask

我们先安装 Flask:

```
(venv)> pip install Flask
```



为节省篇幅,之后“(venv)”这个前缀都省略掉。如无特殊说明,当前目录都是指/home/ubuntu/web_develop。

从 Hello World 开始

我们从最小的应用开始:

```
1 # coding=utf-8
2 from flask import Flask
3
4 app = Flask(__name__)
5
6
7 @app.route('/')
8 def hello_world():
9     return 'Hello World!'
10
11
12 if __name__ == '__main__':
13     app.run(host='0.0.0.0', port=9000)
```

启动它：

```
> python chapter3/section1/hello.py
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
```

打开浏览器，访问“<http://127.0.0.1:9000/>”，就可以看到熟悉的“Hello World!”了。

我们来深入地按行解析这段代码及其背后发生的事情。

- 第 1 行，“# coding=utf-8”是声明 Python 源文件编码的语法。该编码信息后续会被 Python 解析器用于解析源文件。如果没有特殊的原因，应该统一地使用 utf-8，而不要使用 gb18030，gb2312 等类型。为节省篇幅，之后的实例都不再写出这个声明。
- 第 2 行，引入 Flask 类，Flask 类实现了一个 WSGI 应用。
- 第 4 行，app 是 Flask 的实例，它接收包或者模块的名字作为参数，但一般都是传递 `__name__`。让 `flask.helpers.get_root_path` 函数通过传入这个名字确定程序的根目录，以便获得静态文件和模板文件的目录。
- 第 7~9 行，使用 `app.route` 装饰器会将 URL 和执行的视图函数的关系保存到 `app.url_map` 属性上。处理 URL 和视图函数的关系的程序就是路由，这里的视图函数就是 `hello_world`。
- 第 12 行，使用这个判断可以保证当其他文件引用这个文件的时候（例如“`from hello import app`”）不会执行这个判断内的代码，也就是不会执行 `app.run` 函数。
- 第 13 行，执行 `app.run` 就可以启动服务了。默认 Flask 只监听虚拟机的本地 127.0.0.1 这个地址，端口为 5000。而我们对虚拟机做的端口转发端口是 9000，所以需要指定 `host` 和 `port` 参数，0.0.0.0 表示监听所有地址，这样就可以在本机访问了。服务器启动后，会调用 `werkzeug.serving.run_simple` 进入轮询，默认使用单进程单线程的 `werkzeug.serving.BaseWSGIServer` 处理请求，实际上还是使用标准库 `BaseHTTPServer.HTTPServer`，通过 `select.select` 做 0.5 秒的“while True”的事件轮询。当我们访问“<http://127.0.0.1:9000/>”，通过 `app.url_map` 找到注册的“/”这个 URL 模式，就找到了对应的 `hello_world` 函数执行，返回“Hello World!”，状态码为 200。如果访问一个不存在的路径，如访问“<http://127.0.0.1:9000/a>”，Flask 找不到对应的模式，就会向浏览器返回“Not Found”，状态码为 404。

这里需要说明的是，默认的 `app.run` 的启动方式只适合调试，不要在生产环境中使用，生产环境应该使用 Gunicorn 或者 uWSGI。

其他的 `werkzeug` 自带类型还包括 `ThreadedWSGIServer` 和 `ForkingWSGIServer`。



如果想让服务停止，可以发送终止信号或者按 Ctrl-C 键。

配置管理

复杂的项目需要配置各种环境。如果设置项很少，可以直接硬编码进来，比如下面的方式：

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

`app.config` 是 `flask.config.Config` 类的实例，继承自 Python 内置数据结构 `dict`，所以可以使用 `update` 方法：

```
app.config.update(
    DEBUG=True,
    SECRET_KEY='...'
)
```

`app.config` 内置的全部配置变量可以参看 Builtin Configuration Values (<http://bit.ly/28UUgW3>)。如果设置选项很多，想要集中管理设置项，应该将它们存放到一个文件里面。`app.config` 支持多种更新配置的方式。假设现在有个叫作 `settings.py` 的配置文件，其中的内容如下：

```
A = 1
```

可以选择如下三种方式加载。

1. 通过配置文件加载。

```
app.config.from_object('settings') # 通过字符串的模块名字
# 或者引用之后直接传入模块对象
import settings
app.config.from_object(settings)
```

2. 通过文件名字加载。直接传入文件名字，但是不限于只使用.py 后缀的文件名。

```
app.config.from_pyfile('settings.py', silent=True) # 默认当配置文件不存在时
# 会抛出异常，使用silent=True的时候只是返回False，但不会抛出异常
```

3. 通过环境变量加载。这种方式依然支持 `silent` 参数，获得路径后其实还是使用 `from_pyfile` 的方式加载。

```
> export YOURAPPLICATION_SETTINGS='settings.py'
app.config.from_envvar('SETTINGS')
```

调试模式

虽然 `app.run` 这样的方式适用于启动本地的开发服务器，但是每次修改代码后都要手动重启的话，既不方便也不够优雅。如果启用了调试模式，服务器会在代码修改后自动重新载入，并在发生错误时提供一个能获得错误上下文及可执行代码的调试页面。

有两种途径来启用调试模式。

1. 直接在应用对象上设置

```
app.debug = True
app.run()
```

2. 作为 `run` 的参数传入

```
app.run(debug=True)
```

需要注意，开启调试模式会成为一个巨大的安全隐患，因此它绝对不能用于生产环境中。

Werkzeug 从 0.11 版本开始默认启用了 PIN（全称 Personal Identification Number）码的身份验证，旨在让调试环境下的攻击者更难利用调试器。启动程序时可以看到类似的启动提示：

```
> python chapter3/section1/debug.py
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 146-867-947
```

当程序有异常而进入错误堆栈模式，第一次点击某个堆栈想查看对应变量的值的时候，浏览器会弹出一个要求你输入这个 PIN 值的输入框。这个时候需要在输入框中输入 146-867-947，然后确认，Werkzeug 会把这个 PIN 作为 cookie 的一部分存起来（失效时间默认是 8 小时），失效之前不需要重复输入。而这个 PIN 码攻击者是无法知道的。

当然，也可以使用指定 PIN 码的值：

```
> WERKZEUG_DEBUG_PIN=123 python chapter3/section1/debug.py
```

动态 URL 规则

URL 规则可以添加变量部分，也就是将符合同种规则的 URL 抽象成一个 URL 模式，如 `/item/1/`、`/item/2/`、`/item/3/`..... 假如不抽象，我们就得这样写：

```
@app.route('/item/1/')
@app.route('/item/2/')
@app.route('/item/3/')

```

```
def item(id):  
    return 'Item: {}'.format(id)
```

正确的用法是：

```
@app.route('/item/<id>/')  
def item(id):  
    return 'Item: {}'.format(id)
```

尖括号中的内容是动态的，凡是匹配到/item/前缀的 URL 都会被映射到这个路由上，在内部把 id 作为参数而获得。

它使用了特殊的字段标记 <variable_name>，默认类型是字符串。如果需要指定参数类型需要标记成 <converter:variable_name> 这样的格式，converter 有下面几种。

- string：接受任何没有斜杠 “/” 的文本（默认）。
- int：接受整数。
- float：同 int，但是接受浮点数。
- path：和默认的相似，但也接受斜杠。
- uuid：只接受 uuid 字符串。
- any：可以指定多种路径，但是需要传入参数。

```
@app.route('/<any(a, b):page_name>/')
```

访问/a/和访问/b/都符合这个规则，/a/对应的 page_name 就是 a。

如果不希望定制子路径，还可以通过传递参数的方式。比如/people/?name=a，/people/?name=b，这样就可以通过 “name = request.args.get('name')” 获得传入的 name 值。



如果使用 POST 方法，表单参数需要通过 request.form.get('name') 获得。

自定义 URL 转换器

Reddit 可以通过在 URL 中用一个加号（+）隔开各个社区名字，方便同时查看来自多个社区的帖子。比如访问 “http://reddit.com/r/flask+lisp”，就可以同时看 flask 和 lisp 两个社区的帖子。我们自定义一个转换器来实现这个功能，它还可以设置所使用的分隔符，不一定要用加号 “+”。

```
import urllib

from flask import Flask
from werkzeug.routing import BaseConverter

app = Flask(__name__)

class ListConverter(BaseConverter):

    def __init__(self, url_map, separator='+'):
        super(ListConverter, self).__init__(url_map)
        self.separator = urllib.unquote(separator)

    def to_python(self, value):
        return value.split(self.separator)

    def to_url(self, values):
        return self.separator.join(BaseConverter.to_url(value)
                                   for value in values)

app.url_map.converters['list'] = ListConverter

@app.route('/list1/<list:page_names>/')
def list1(page_names):
    return 'Separator: {} {}'.format('+', page_names)

@app.route('/list2/<list(separator=u"|"):page_names>/')
def list2(page_names):
    return 'Separator: {} {}'.format('|', page_names)
```

这样我们访问 “/list1/a+b/” 和 “/list2/a|b/” 就能实现同样的功能了。自定义转换器需要继承至 `BaseConverter`，要设置 `to_python` 和 `to_url` 两个方法。

- `to_python`：把路径转换成一个 Python 对象。
- `to_url`：把参数转换成为符合 URL 的形式。

HTTP 方法

HTTP 有多个访问 URL 方法，默认情况下，路由只回应 GET 请求，但是通过 `app.route` 装饰器传递 `methods` 参数可以改变这个行为：

```
@app.route('/login', methods=['GET', 'POST'])
@app.route('/j/item/<id>', methods=['DELETE', 'POST'])
```

如果存在 GET，那么也会自动地添加 HEAD 方法，无须干预。它会确保遵照 HTTP RFC（描述 HTTP 协议的文档）（<http://bit.ly/2932IiA>）处理 HEAD 请求，所以你完全可以忽略这部分的 HTTP 规范。从 Flask 0.6 起，它也实现了 OPTIONS 的自动处理。

下面简要介绍 HTTP 方法和使用场景。

- GET：获取资源，GET 操作应该是幂等的。
- HEAD：想要获取信息，但是只关心消息头。应用应该像处理 GET 请求一样来处理它，但是不返回实际内容。
- POST：创建一个新的资源。
- PUT：完整地替换资源或者创建资源。PUT 操作虽然有副作用，但应该是幂等的。
- DELETE：删除资源。DELETE 操作有副作用，但也是幂等的。
- OPTIONS：获取资源支持的所有 HTTP 方法。
- PATCH：局部更新，修改某个已有的资源。



幂等表示在相同的数据和参数下，执行一次或多次产生的效果是一样的。

唯一 URL

Flask 的 URL 规则基于 Werkzeug 的路由模块。这个模块背后的思想是基于 Apache 以及更早的 HTTP 服务器的主张，希望保证优雅且唯一的 URL。

举个例子：

```
@app.route('/projects/')
def projects():
    return 'The project page'
```

上述例子很像一个文件系统中的文件夹，访问一个结尾不带斜线的 URL 会被重定向到带斜线的规范的 URL 上去，这样也有助于避免搜索引擎索引同一个页面两次。

再看一个例子：

```
@app.route('/about')
def about():
    return 'The about page'
```

URL 结尾不带斜线，很像文件的路径，但是当访问带斜线的 URL（/about/）会产生一个 404 “Not Found” 错误。

构造 URL

用 `url_for` 构建 URL，它接受函数名作为第一个参数，也接受对应 URL 规则的变量部分的命名参数，未知的变量部分会添加到 URL 末尾作为查询参数。构建 URL 而不选择直接在代码中拼 URL 的原因有两点：在未来有更改的时候只需要一次性修改 URL，而不用到处去替换；URL 构建会转义特殊字符和 Unicode 数据，这些工作不需要我们自己处理。

感受下面这个例子：

```
from flask import Flask, url_for
app = Flask(__name__)
```

```
@app.route('/item/1/')
def item(id):
    pass
```

```
with app.test_request_context():
    print url_for('item', id='1')
    print url_for('item', id=2, next='/')
```

`test_request_context` 帮助我们在交互模式下产生请求上下文。

执行它：

```
> python chapter3/section1/url.py
/item/1/?id=1
/item/1/?id=2&next=%2F
```

跳转和重定向

跳转（状态码 301）多用于旧网址在废弃前转向新网址以保证用户的访问，有页面被永久性移走的概念。重定向（状态码 302）表示页面是暂时性的转移。但是也不建议经常性使用重定向。在 Flask 中它们都是通过 `flask.redirect` 实现的：

```
redirect(location) # 默认是302
redirect(location, code=301) # 通过code参数可以指定状态码
```

Flask 还支持 303、305、307 重定向，但是较少被用到。

基于前面所讲的内容，我们来看一个更全面的例子。首先是存放配置的 `config.py`:

```
DEBUG = False
```

```
try:
    from local_settings import *
except ImportError:
    pass
```

`local_settings.py` 文件是可选存在的，它不进入版本库。这是常用的通过本地配置文件重载版本库配置的方式。

基于上面所讲的内容，我们看一个更复杂的应用 (`simple.py`):

```
1 from flask import Flask, request, abort, redirect, url_for
2
3 app = Flask(__name__)
4 app.config.from_object('config')
5
6
7 @app.route('/people/')
8 def people():
9     name = request.args.get('name')
10    if not name:
11        return redirect(url_for('login'))
12    user_agent = request.headers.get('User-Agent')
13    return 'Name: {0}; UA: {1}'.format(name, user_agent)
14
15
16 @app.route('/login/', methods=['GET', 'POST'])
17 def login():
18     if request.method == 'POST':
19         user_id = request.form.get('user_id')
20         return 'User: {} login'.format(user_id)
21     else:
22         return 'Open Login page'
23
24
25 @app.route('/secret/')
26 def secret():
27     abort(401)
28     print 'This is never executed'
29
30
31 if __name__ == '__main__':
32     app.run(host='0.0.0.0', port=9000, debug=app.debug)
```

这个例子有如下细节。

- 第 7 行，访问/people 的请求会被 301 跳转到/people/上，保证了 URL 的唯一性。
- 第 12 行，request.headers 存放了请求的头信息，通过它可以获取 UA 值。
- 第 18 行，request.method 的值就是请求的类型。
- 第 27 行，执行 abort(401) 会放弃请求并返回错误代码 401，表示禁止访问。之后的语句永远不会被执行。
- 第 32 行，能使用 debug=app.debug 是因为 flask.config.ConfigAttribute 在 app 中做了配置的代理，目前存在的配置代理项有：

```
app.debug -> DEBUG
app.testing -> TESTING
app.secret_key -> SECRET_KEY
app.session_cookie_name -> SESSION_COOKIE_NAME
app.permanent_session_lifetime -> PERMANENT_SESSION_LIFETIME
app.use_x_sendfile -> USE_X_SENDFILE
app.logger_name -> LOGGER_NAME
```

上面例子中的 app.debug 其实就是 app.config['DEBUG']。

响应

视图函数的返回值会被自动转换为一个响应对象，转换的逻辑如下：

- 如果返回的是一个合法的响应对象，它会从视图直接返回。
- 如果返回的是一个字符串，会用字符串数据和默认参数创建以字符串为主体，状态码为 200，MIME 类型是 text/html 的 werkzeug.wrappers.Response 响应对象。
- 如果返回的是一个元组，且元组中的元素可以提供额外的信息。这样的元组必须是 (response, status, headers) 的形式，但是需要至少包含一个元素。status 值会覆盖状态代码，headers 可以是一个列表或字典，作为额外的消息头。
- 如果上述条件均不满足，Flask 会假设返回值是一个合法的 WSGI 应用程序，并通过 Response.force_type(rv, request.environ) 转换为一个请求对象。

下面的视图函数：

```
@app.errorhandler(404)
def not_found(error):
    return render_template('error.html'), 404
```


可以改成如下显式地调用 `make_response` 的方式：

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    return resp
```

第二种方法很灵活，可以添加一些额外的工作，比如设置 cookie、头信息等。

API 都是返回 JSON 格式的响应，需要包装 `jsonify`。可以抽象一下，让 Flask 自动帮我们做这些工作（`app_response.py`）：

```
from flask import Flask, jsonify
from werkzeug.wrappers import Response
app = Flask(__name__)

class JSONResponse(Response):
    @classmethod
    def force_type(cls, rv, environ=None):
        if isinstance(rv, dict):
            rv = jsonify(rv)
        return super(JSONResponse, cls).force_type(rv, environ)

app.response_class = JSONResponse
```

```
@app.route('/')
def hello_world():
    return {'message': 'Hello World!'}

@app.route('/custom_headers')
def headers():
    return {'headers': [1, 2, 3]}, 201, [('X-Request-Id', '100')]

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

启动它之后，就可以在另一个终端看看自定义头信息的效果了。看效果之前，先安装 `httpie`（<https://github.com/jkbrzt/httpie>）：

```
> pip install httpie
```

`httpie` 是一个使用 Python 编写的，提供了语法高亮、JSON 支持，可以替代 `curl` 的工具，它也可方便地集成到 Python 项目中。本书绝大多数命令行下请求数据时都使用它。

现在请求/custom_headers:

```
> http http://0.0.0.0:9000/custom_headers
HTTP/1.0 201 CREATED
Content-Length: 44
Content-Type: application/json
Date: Thu, 26 May 2016 17:34:54 GMT
Server: Werkzeug/0.11.10 Python/2.7.11+
X-Request-Id: 100
```

```
{
  "headers": [
    1,
    2,
    3
  ]
}
```

视图中也可以直接指定状态字符串，如使用'201 CREATED' 替代数字的 201。

静态文件管理

Web 应用大多会提供静态文件服务以便给用户更好的访问体验。静态文件主要包含 CSS 样式文件、JavaScript 脚本文件、图片文件和字体文件等静态资源。Flask 也支持静态文件访问，默认只需要在项目根目录下创建名字为 static 的目录，在应用中使用“/static”开头的路径就可以访问。但是为了获得更好的处理能力，推荐使用 Nginx 或者其他 Web 服务器管理静态文件。

不要直接在模板中写死静态文件路径，应该使用 url_for 生成路径。举个例子：

```
url_for('static', filename='style.css')
```

生成的路径就是“/static/style.css”。当然，我们也可以定制静态文件的真实目录：

```
app = Flask(__name__, static_folder='/tmp')
```

那么访问“http://localhost:9000/static/style.css”，也就是访问/tmp/style.css 这个文件。

即插视图

即插视图的灵感来自 Django 的基于类而不是函数的通用视图方式，这样的视图就可以支持继承了。视图类型有两种类型。

标准视图

标准视图需要继承 `flask.views.View`，必须实现 `dispatch_request`。看一个例子（`app_view.py`）：

```
from flask import Flask, request, render_template
from flask.views import View

app = Flask(__name__, template_folder='.././templates')

class BaseView(View):
    def get_template_name(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        if request.method != 'GET':
            return 'UNSUPPORTED!'

        context = {'users': self.get_users()}
        return self.render_template(context)

class UserView(BaseView):

    def get_template_name(self):
        return 'chapter3/section1/users.html'

    def get_users(self):
        return [{
            'username': 'fake',
            'avatar': 'http://lorempixel.com/100/100/nature/'
        }]

app.add_url_rule('/users', view_func=UserView.as_view('userview'))

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

模板存放在 `~/web_develop/templates` 下，使用 `__name__` 来获取模板目录，`template_folder` 是相对于 `app.py` 文件的，需要设置成 `'.././templates'` 才能找到正确的模板目录。

基于调度方法的视图

`flask.views.MethodView` 对每个 HTTP 方法执行不同的函数（映射到对应方法的小写的同名方法上），这对 RESTful API 尤其有用。看一个例子（`app_api.py`）：

```
from flask import Flask, jsonify
from flask.views import MethodView

app = Flask(__name__)

class UserAPI(MethodView):

    def get(self):
        return jsonify({
            'username': 'fake',
            'avatar': 'http://lorempixel.com/100/100/nature/'
        })

    def post(self):
        return 'UNSUPPORTED!'

app.add_url_rule('/user', view_func=UserAPI.as_view('userview'))

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

通过装饰 `as_view` 的返回值来实现对于视图的装饰功能，常用于权限的检查、登录验证等：

```
def user_required(f):
    def decorator(*args, **kwargs):
        if not g.user:
            abort(401)
        return f(*args, **kwargs)
    return decorator

view = user_required(UserAPI.as_view('users'))
app.add_url_rule('/users/', view_func=view)
```

从 Flask 0.8 开始，还可以通过在继承 `MethodView` 的类中添加 `decorators` 属性来实现对视图的装饰：

```
class UserAPI(MethodView):
    decorators = [user_required]
```

蓝图

蓝图（Blueprint）实现了应用的模块化，使用蓝图让应用层次清晰，开发者可以更容易的开发和维护项目。蓝图通常作用于相同的 URL 前缀，比如/user/:id、/user/profile 这样的地址，都以/user 开头，那么它们就可以放在一个模块中。看一个最简单的示例（user.py）：

```
from flask import Blueprint

bp = Blueprint('user', __name__, url_prefix='/user')

@bp.route('/')
def index():
    return 'User"s Index page'
```

每个模块都会暴露一个全局变量 bp。再看主程序（app_bp.py）：

```
from flask import Flask
import user

app = Flask(__name__)
app.register_blueprint(user.bp)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

使用 register_blueprint 注册模块，如果想去掉模块只需要去掉对应的注册语句即可。

子域名

现在许多 SaaS 应用为用户提供一个子域名来访问，可以借助 subdomain 来实现同样的功能（app_subdomain.py）：

```
from flask import Flask, g

app = Flask(__name__)
app.config['SERVER_NAME'] = 'example.com:9000'

@app.url_value_preprocessor
def get_site(endpoint, values):
    g.site = values.pop('subdomain')
```

```
@app.route('/', subdomain='<subdomain>')
def index():
    return g.site

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

在虚拟机上绑定一下域名，也就是在/etc/hosts 添加一行：

```
127.0.0.1 a.example.com b.example.com
```

现在验证它：

```
> http http://b.example.com:9000 --print b # b表示只输出响应的主体
b
```

命令行接口

在 Flask 0.11 之前，启动的应用的端口、主机地址以及是否开启 DEBUG 模式，都需要在代码中明确指定，一个比较好的方式是使用第三方扩展 Flask-Script 管理。从 Flask 0.11 开始，Flask 集成了 Click，现在可以直接在命令行直接执行 flask 命令启动 Flask 应用了：

```
> export FLASK_APP=chapter3/section1/hello.py
> export FLASK_DEBUG=1
> flask run -h 0.0.0.0 -p 9000
```

这种命令行启动的方式要灵活得多，命令 flask 还支持 shell 子命令：

```
> flask shell
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
App: hello [debug]
Instance: /home/ubuntu/web_develop/chapter3/section1/instance
>>> app
<Flask 'hello'>
```

新的方案甚至能替换 Flask-Script。现在我们基于 flask.cli 模块添加两个子命令（app_cli.py）。

第一个子命令 initdb，用来初始化数据库，这里没有实际的逻辑，只为了演示 click 的使用方法：

```
import click
from flask import Flask

app = Flask(__name__)
```

```
@app.cli.command()
def initdb():
    click.echo('Init the db')
```

先指定 FLASK_APP 变量：

```
> export FLASK_APP=chapter3/section1/app_cli.py
```

现在就可以使用 initdb 了：

```
> flask initdb
Init the db
```

再复杂些，实现一个叫作 new_shell 的子命令，它使用 IPython 环境（除非没有安装 IPython 或者强制要求使用默认的交互环境）：

```
from flask.cli import with_appcontext
```

```
try:
    import IPython # noqa
    has_ipython = True
except ImportError:
    has_ipython = False
```

```
def plain_shell(user_ns, banner):
    sys.exit(code.interact(banner=banner, local=user_ns))
```

```
def ipython_shell(user_ns, banner):
    IPython.embed(banner1=banner, user_ns=user_ns)
```

```
@app.cli.command('new_shell', short_help='Runs a shell in the app context.')
@click.option('--plain', help='Use Plain Shell', is_flag=True)
@with_appcontext
def shell_command(plain):
    from flask.globals import _app_ctx_stack
    app = _app_ctx_stack.top.app
    banner = 'Python %s on %s\nApp: %s\nInstance: %s' % (
        sys.version,
        sys.platform,
        app.import_name,
        app.debug and ' [debug]' or '',
        app.instance_path,
```

```

)
user_ns = app.make_shell_context()
use_plain_shell = not has_ipython or plain
if use_plain_shell:
    plain_shell(user_ns, banner)
else:
    ipython_shell(user_ns, banner)

```

其中 `app.cli.command` 用来指定子命令的名字和帮助信息，`click.option` 给予命令添加参数，由于 `new_shell` 需要使用 `app` 这个上下文，所以需要添加 `with_appcontext` 这个装饰器。

模板

在之前的章节中，视图函数直接返回文本，而在实际生产环境中其实很少这样用，因为实际的页面大多是带有样式和复杂逻辑的 HTML 代码，这可以让浏览器渲染出非常漂亮和复杂的效果。页面内容应该是可重用的，而且需要执行更高级的功能。首先看一下 Python 自带的模板 `string.Template`：

```

In : from string import Template
In : s = Template('$who likes $what')
In : s.substitute(who='tim', what='kung pao') # 按照给定的参数插入到对应的变量上
Out: 'tim likes kung pao'
In : s = Template("$var is here but $missing is not provided")
In : s.safe_substitute(var='tim')
Out: 'tim is here but $missing is not provided'
In : class MyTemplate(Template):
....:     delimiter = '@' # 使用@为分隔符
....:     idpattern = '[a-z]+\.[a-z]+' # 符合的模式才会被替换
....:
In : t = MyTemplate('@with.dot @notdoted')
In : t.safe_substitute({'with.dot': 'replaced', 'notdoted': 'not replaced'})
Out: 'replaced @notdoted'

```

自带的模板提供的功能大抵如此，支持很有限：不能写控制语句，无法继承重用。这对于 Web 开发来说远远不够，需要使用第三方的模板系统。目前市面上有非常多的模板系统，其中最知名的就是 Jinja2 和 Mako。本节我们将分别介绍它们。

Jinja2

Jinja 是日本寺庙的意思，并且寺庙的英文 `temple` 和 `template` 的发音类似。Jinja2 是 Flask 默认的仿 Django 模板的一个模板引擎，由 Flask 的作者开发。它速度快，被广泛使用，并且提供了可选的沙箱模板来保证执行环境的安全。它有如下优点：

- 让 HTML 设计者和后端 Python 开发工作分离。
- 减少使用 Python 的复杂程度，页面逻辑应该独立于业务逻辑，这样才能开发出易于维护的程序。
- 模板非常灵活、快速和安全，对设计者和开发者会更友好。
- 提供了控制语句、继承等高级功能，减少开发的复杂度。

Jinja2 是 Flask 的一个依赖，因为我们之前已经安装了 Flask，所以 Jinja2 也随之安装了。否则可以单独安装：

```
> pip install Jinja2
```



Jinja2 从 2.7 开始已经依赖 MarkupSafe 了，MarkupSafe 的 C 实现要快得多，使用 pip 安装 Jinja2 时会自动安装它。如果不方便升级到新版本的 Jinja2，但当前版本大于 2.5.1，可手动安装 MarkupSafe。

API 的基本使用方式

Jinja2 通过 Template 类创建并渲染模板：

```
In : from jinja2 import Template
In : template = Template('Hello {{ name }}!')
In : template.render(name='Xiao Ming')
Out: u'Hello Xiao Ming!'
```

是不是和 string.Template 做的事情很像呢？上面的代码片段背后的逻辑大致是这样的：

```
In : from jinja2 import Environment
In : env = Environment()
In : template = env.from_string('Hello {{ name }}!')
In : template.render(name='Xiao Ming')
Out: u'Hello Xiao Ming!'
```

Environment 的实例用于存储配置和全局对象，然后从文件系统或其他位置加载模板：

```
> echo "Hello {{ name }}" > templates/chapter3/section2/jinja2/hello.html
> touch app.py
> ipython
In : from jinja2 import Environment, PackageLoader
In : env = Environment(loader=PackageLoader('app', 'templates/chapter3/section2/jinja2'))
In : template = env.get_template('hello.html')
In : template.render(name='Xiao Ming')
Out: u'Hello Xiao Ming'
```

通过 Environment 创建了一个模板环境，模板加载器（loader）会在 templates 文件夹中寻找模板。因为这里是测试，所以只是用到 app.py 这个空文件作为包名。由于模板文件在模板目录的子目录下，也可以这样获取：

```
env = Environment(loader=PackageLoader('app', 'templates'))
template = env.get_template('chapter3/section2/jinja2/hello.html')
```

使用模板加载器的另一个明显的好处就是可以支持模板继承。

Jinja2 的基本语法

模板仅仅是文本文件，它可以使用任何基于文本的格式（HTML、XML、CSV、LaTeX 等），它并没有特定的扩展名，通常使用.html 作为后缀名。模板包含“变量”或“表达式”，这两者在模板求值的时候会被替换为值。模板中还有标签和控制语句。

下面是一个简单的模板（simple.html）：

```
1 <!DOCTYPE html>
2 <html lang="en">
3     <head>
4         <title>Simple Page</title>
5     </head>
6     <body>
7         {# This is a Comment #}
8         <ul id="navigation">
9             {% for item in items %}
10                 <li><a href="{{ item.href }}">{{ item['caption'] }}</a></li>
11             {% endfor %}
12         </ul>
13
14         <h1>{{ title | trim }}</h1>
15         <p>{{ content }}</p>
16
17     </body>
18 </html>
```

我们来解析下这个模板的语法。

- 第 1 行，声明文档类型是 HTML 5。
- 第 7、9、10 行，这是三种分隔符，每种分隔符都包含开始标记和结束标记。
 - ◊ {# ... #}：模板注释。它不会出现在渲染的页面里。
 - ◊ {% ... %}：用于执行诸如 for 循环或赋值的语句。
 - ◊ {{ ... }}：用于把表达式的结果输出到模板上。

- 第9行，出现了“for 循环”这种控制结构。语法是 `{% for X in Y %} ... {% endfor %}`，控制语句都需要以 `endxxx` 作为结束。
- 第10行，应用把变量传递到模板，可以使用点（.）来访问变量的属性，也可以使用括号语法（[]）。下面的两行效果几乎是一样的：

```
{{ item.href }}  
{{ item['href'] }}
```

- 第14行，`trim` 是一个过滤器，在模板中通过管道符号（|）把变量和过滤器分开。也可以使用多个过滤器，如 `{{ title|trim|striptags }}`，`striptags` 也是一个过滤器。Jinja2 内置了非常多的过滤器，全部过滤器可以在 <http://bit.ly/29RZ1fK> 找到，一定要熟悉这些过滤器，它们大多都很常用。

模板继承

合理使用模板继承，让模板能重用，能提高工作效率和代码质量。

首先定义一个基础的“骨架”模板（`base.html`）：

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    {% block head %}  
      <link rel="stylesheet" href="style.css" />  
      <title>{% block title %}{% endblock %} - My Webpage</title>  
    {% endblock %}  
  </head>  
  <body>  
    <div id="content">  
      {% block content %}  
      {% endblock %}  
    </div>  
    <div id="footer">  
      {% block footer %}  
      {% endblock %}  
    </div>  
  </body>  
</html>
```

这个模板有如下细节：

- “`{% block XXX %} ... {% endblock %}`” 是一个代码块，可以在子模板重载。
- `head` 的代码块有默认内容，而 `content` 和 `footer` 都是没有内容的。这3个块需要在子模板中被重载，如果子模板没有重载，就用这个基类模板的定义显示默认内容。

接着看子模板 (index.html):

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
<h1>Index</h1>
<p class="important">
    Welcome on my awesome homepage.
</p>
{% endblock content %}
```

子模板有如下细节:

- index.html 继承了 base.html 里面的内容。extends 标签应该在模板中一开始就使用。
- 标题被重载, 换成了 “Index”。
- head 块被重载。但是首先使用了 super(), 表示先使用 base.html 的 head 块的内容, 再基于此添加 CSS 样式。
- content 块被重载, 其中在块的结束标签中加入了名称, 这样可以改善模板的可读性。
- footer 没有被重载, 什么都不显示。

如果你想要多次使用一个块, 可以使用特殊的 “self” 变量并调用与块同名的函数:

```
<title>{% block title %}{% endblock %}</title>
<h1>{{ self.title() }}</h1>
```

宏

宏类似常规编程语言中的函数。它用于把常用行为抽象成可重用的函数:

```
In : Template('''
...: {% macro hello(name) %}
...:     Hello {{ name }}
...: {% endmacro %}
...: <p>{{ hello('world') }}</p>
...: ''').render()
Out: u'\n\n<p>\n    Hello world\n</p>'
```

可以像函数一样调用宏。

赋值

在代码块中，你也可以为变量赋值。赋值使用 `set` 标签，并且可以为多个变量赋值：

```
In : print Template('''
...: {% set a = 1 %}
...: {% set b, c = range(2) %}
...: <p>{{ a }} {{ b }} {{ c }}</p>
...: ''').render()
Out: u'\n\n\n<p>1 0 1</p>'
```

include

`include` 语句用于包含一个模板，渲染时会在 `include` 语句的对应位置添加被包含的模板内容：

```
{% include 'header.html' %}
    Body
{% include 'footer.html' %}
```

`include` 可以使用 “`ignore missing`” 标记，如果模板不存在，Jinja 会忽略这条语句：

```
{% include "sidebar.html" ignore missing %}
```

import

Jinja2 支持在不同的模板中导入宏并使用，与 Python 中的 `import` 语句类似。有两种方式来导入模板：可以把整个模板导入到一个变量（`import xx`）或从其中导入特定的宏（`from xx import yy`）。

现在有一个宏模板（`macro.html`）：

```
{% macro hello(name) %}
    Hello {{ name }}
{% endmacro %}

{% macro strftime(time, fmt='%Y-%m-%d %H:%M:%S') %}
    {{ time.strftime(fmt) }}
{% endmacro %}
```

引用并调用宏的方法如下（`hello_macro.html`）：

```
{% import 'macro.html' as macro %}
{% from 'macro.html' import hello as _hello, strftime %}

<p>{{ macro.hello('world') }}</p>
<p>{{ strftime(time) }}</p>
```

看一下渲染的效果：

```
In : from jinja2 import FileSystemLoader, Environment
In : from datetime import datetime
In : loader = FileSystemLoader('/home/ubuntu/web_develop/templates/chapter3/section2/
    jinja2')
In : template = Environment(loader=loader).get_template('hello_macro.html')
In : print template.render(time=datetime.now())
```

```
<p>
    Hello world
</p>
<p>
    2016-05-27 04:37:11
</p>
```

Mako

Mako 是另一个知名模板语言。它从 Django、Jinja2、Genshi 等模板借鉴了很多语法和 API。它有如下优点：

- 性能和 Jinja2 相近，这一点 Jinja2 也承认（<http://bit.ly/28RvFiM>）。
- 有大型网站在使用，有质量保证。Reddit 在 2011 年的月 PV 就达到 10 亿，豆瓣几乎全部用户产品都使用 Mako 模板，所以不需要担心没有大公司使用的案例。
- 有知名 Web 框架支持。Pylons 和 Pyramid 这两个 Web 框架内置 Mako，而且把它作为默认模板。
- 支持在模板中写几乎原生的 Python 语法的代码，对 Python 工程师友好。
- 自带完整的缓存系统。Mako 提供非常好的扩展接口，很容易切换成其他的缓存系统。

Jinja2 和 Mako 的设计哲学有一点不同：Jinja2 认为应该尽可能把逻辑从模板中移除，界限清晰，不允许在模板内写 Python 代码，也不支持全部的 Python 内置函数（只提供了很有限、最常用的一部分）；而 Mako 正好相反，它最后会编译成 Python 代码以达到性能最优，在模板里面可以自由写后端逻辑，不需要传递就可以使用 Python 自带的数据结构和内置类。Jinja2 带来的好处是模板引擎易于维护，并且模板有更好的可读性；而 Mako 是一个对 Python 工程师非常友好的语言，限制很少，完成模板开发工作时更有效率，整个项目的代码可维护性更好。

我们先安装它：

```
> pip install Mako
```

基本 API 的使用

Mako 也可以通过 Template 类创建一个模板实例并渲染它：

```
In : from mako.template import Template
In : template = Template('Hello ${name}!')
In : template.render(name='Xiao Ming')
Out: u'Hello Xiao Ming!'
```

Mako 的变量使用了 “\${...}” 的风格。

模板文件后缀不强制以 “.mako” 结尾，使用 “.html” 甚至 “.txt” 都是可以接受的。

```
In : Template(filename='templates/chapter3/section2/mako/hello.mako').render(name='
    XiaoMing')
Out: u'Hello XiaoMing\n'
```

可以优化一下性能，保存编译后的模板，下次有参数相同的调用就能使用缓存的结果：

```
In : Template(filename='templates/chapter3/section2/mako/hello.mako', module_directory
    = '/tmp/mako_cache').render(name='XiaoMing')
Out: u'Hello XiaoMing\n'
```

看一下生成的 Mako 文件：

```
> tree /tmp/mako_cache
/tmp/mako_cache
├── templates
│   ├── chapter3
│   │   └── section2
│   │       └── mako
│   │           ├── hello.mako.py
│   │           └── hello.mako.pyc
```

缓存的文件是按照模板路径的结构保存的。

使用 TemplateLookup

上面的例子只是单个模板文件的渲染。试想下如果有模板继承，或者模板引用了其他模板，如何告诉 Mako 模板的搜索路径都有哪些呢？TemplateLookup 就是做这件事的：

```
In : from mako.lookup import TemplateLookup
In : mylookup = TemplateLookup(directories=['templates/chapter3/section2/mako'])
In : template = Template('<%include file="hello.mako"/>', lookup=mylookup)
In : template.render(name='XiaoMing')
Out: u'Hello XiaoMing\n'
```

和 Jinja2 的用法更相近的方式是：

```
In : mylookup.get_template('hello.mako').render(name='XiaoMing')
Out: u'Hello XiaoMing\n'
```

TemplateLookup 也支持 module_directory 参数：

```
mylookup = TemplateLookup(directories=['templates/chapter3/section2/mako'],
    module_directory='/tmp/mako_modules')
```

渲染模板还能使用相对路径：

```
In : mylookup.get_template('/hello.mako').render(name='XiaoMing')
Out: u'Hello XiaoMing\n'
```

模板名字使用 “/” 开头，这是因为搜索目录包含了 templates/chapter3/section2/mako，这个 “/” 只是表示相对的路径。

Mako 的基本语法

下面是一个简单的模板（simple.html）：

```
1 <%!
2     from datetime import datetime
3     from itertools import repeat
4 %>
5
6 <%inherit file="base.html"/>
7 <%namespace name="utils" file="/utils.html" />
8
9 <%
10     rows = repeat(range(10), 10)
11 %>
12
13 <%include file="/nav.html" />
14
15
16 <%def name="main()">
17     ## this is a comment
18     <h2>${utils.strftime(datetime.now())}</h2>
19
20     <table>
21         % for row in rows:
22             ${makerow(row)}
23         % endfor
24     </table>
25 </%def>
```



```
26
27 <%def name="makerow(row)">
28     <tr>
29         % for name in row:
30             <td>${name}</td>
31         % endfor
32     </tr>
33 </%def>
```

此模板有如下细节：

- 第 1~4 行，“<%! ... %>”是模块级别的块元素，常用来引入模块、声明全局变量和全局函数等。
- 第 6 行，表示这个模板继承了 base.html。
- 第 7 行，表示把 /utils.html 当作一个模块一样声明，使用“utils”这个命令空间。之后可以直接调用 utils 里面的函数、变量等。
- 第 9~11 行，“<% ... %>”表示 Python 代码块，在里面可以自由地写 Python 代码，但是它不是全局的，建议在函数内使用，保证在调用模板函数的时候可以访问这些定义的代码。
- 第 13 行，表示直接把“/nav.html”这个模板的内容插入到当前位置。
- 第 16、27 行，“<%def name=”main()”>”和“<%def name=”makerow(row)”>”是两个函数。区别是 makerow 需要传入参数。
- 第 17 行，以“##”开头的行表示一个注释。如果有多行注释，也可以使用“<%doc> ... </%doc>”的方式。
- 第 18 行，“\${ ... }”中要执行的可以是模板中的函数，也可以是 Python 代码，执行结果就直接输出到 HTML 页面上了。
- 第 21 行，使用“%”标签就可以直接写 Python 语法的控制语句了。“% for row in rows:”是一个 for 循环语句，它用 Python 代码风格生成 HTML 代码。

<%page>

上面例子中的“<%include ... >”并没有带参数，nav.html 里面只要用“<%page />”即可，当需要传输参数的时候这样使用：

```
<%page args="x, y, z='default'"/>
```

插入模板的语法如下：

```
<%namespace name="utils" file="/utils.html" args="1, 2, z='z'" />
```

“<%page>”还可以指定缓存方式：

```
<%page cached="True" cache_type="memory"/>
```

<%block>

“%block”和“%def”很像，它受 Jinja2 的 block 启发，在定义的地方被渲染，无须像“%def”那样当需要调用时才会被渲染。“%block”也可以接收缓存、过滤器的参数：

```
<html>
  <body>
    <%block cached="True" cache_timeout="60">
      This content will be cached for 60 seconds.
    </%block>
  </body>
</html>
```

我们也可以给块加个名字以便重复调用：

```
<div name="page">
  <%block name="pagecontrol">
    <a href="#">previous page</a> |
    <a href="#">next page</a>
  </%block>

  <table>
    ## some content
  </table>

  ${pagecontrol()}
</div>
```

pagecontrol 共渲染了两次。

<%namespace>

“<%namespace>”的作用很像 Python 的 import，可以把其他模板当成 Python 模块一样引用进来：

```
<%namespace file="/utils.html" import="strftime"/>
```

这样就可以直接使用 strftime 了：

```
<h2>${strftime(datetime.now())}</h2>
```



import 支持 “*” 操作符（可能会影响性能，建议采用显式的 import）：

```
<%namespace file="/utils.html" import="*" />
```

file 参数还可以接收表达式，动态地传入文件名：

```
<%namespace name="dyn" file="${context['namespace_name']}" />
```

除了上面通过 “utils.strftime” 的方式调用，还可用以下两种方式调用：

```
<%utils:strftime args='${datetime.now()}' />
<%call expr='utils.strftime()' args='${datetime.now()}' ></%call>
```

过滤器

Mako 模板中同样使用管道符号（|）把过滤器和变量分隔开，但需要注意的是，多个过滤器是用逗号隔开的：

```
${ "this is some text" | u }
${ "<tag>some value</tag>" | h,trim }
```

Mako 中的常用过滤器包含如下 4 种。

- u：URL 的转换，等价于 “urllib.quote_plus(string.encode('utf-8'))”。

```
In : Template('Hello ${ name|u }!').render(name=u'中文')
Out: u'Hello %E4%B8%AD%E6%96%87!'
```

- h：HTML 转换，等价于 “markupsafe.escape(string)”。如果没有安装 markupsafe 模块的话，等价于 “cgi.escape(string, True)”。

```
In : Template('Hello ${ name|h }!').render(name='<div>n</div>')
Out: u'Hello &lt;div&gt;n&lt;/div&gt;!'
```

- trim：过滤行首和行尾的空格，实际上是 “string.strip()”。

```
In : Template('Hello ${ name|trim }!').render(name=' a ')
Out: u'Hello a!'
```

- n：禁用默认的过滤器。

在 Mako 中定义一个过滤器非常简单（my_filters.html）：

```
<%!
def div(text):
    return "<div>" + text + "</div>"
```

```
%>
```

```
Here's a div: ${ "text" | div }
```

函数 `div` 需要输入一个参数，参数就是过滤之前的文本，也就是文本：“text”。

可以使用 `default_filters` 参数指定全局的设置。如果不指定，在 Python 2 中，默认设置是 “[‘unicode’]”，在 Python 3 中是 “[‘str’]”：

```
mylookup = TemplateLookup(directories=['templates/chapter3/section2/mako'],
    default_filters=['unicode', 'h', 'decode.utf8'])
```

这样就不需要在页面指定 “`expression_filter='h'`” 了。如果想要全局开启自定义的过滤器，需要使用如下方式：

```
mylookup = TemplateLookup(
    directories=['templates/chapter3/section2/mako'],
    default_filters=['str', 'myfilter'],
    imports=['from mypackage import myfilter'])
```

Mako 模板编译完成后会生成这样的代码（截取相关的一部分）：

```
from mypackage import myfilter

def render_body(context, **pageargs):
    name = context.get('name', UNDEFINED)
    __M_writer = context.writer()
    __M_writer(u'Hello ')
    __M_writer(myfilter(str(name)))
    __M_writer(u'\n')
```

`render_body` 是每个 Mako 模板编译完成之后生成的 Python 代码的主函数，渲染的时候就是通过调用它生成 HTML 代码的。过滤器的执行顺序和当时指定的顺序是一样的。

模板继承

我们来使用 Mako 的模板继承，生成和 Jinja2 一样的 HTML 页面。首先看一下骨架模板（`base.html`）：

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <%block name="head">
      <link rel="stylesheet" href="style.css" />
      <title>${ title() } - My Webpage</title>
    </%block>
```

```
</head>
<body>
  <div id="content">
    <%block name="content"/>
  </div>
  <div id="footer">
    <%block name="footer"/>
  </div>
</body>
</html>

<%def name="title()">
</%def>
```

和 Jinja2 的模板相比，只是语法不同。再看一下子模板（index.html）：

```
<%inherit file="base.html"/>

<%def name="title()">Index</%def>

<%block name="head">
  ${ parent.head() }
  <style type="text/css">
    .important { color: #336699; }
  </style>
</%block>

<%block name="content">
  <h1>Index</h1>
  <p class="important">
    Welcome on my awesome homepage.
  </p>
</%block>
```



title 当然也可以使用“%block”来实现。模板中也能多次调用同一个函数，使用“self.<defname>”的方式，比如 title 可以使用“\${ self.title() }”。

Mako 排错

Mako 有一个让人非常迷惑的地方，即出现 `NameError (NameError: Undefined)`，尤其是在使用了模板多继承的情况下。举个例子，不小心在模板中使用了一个没有被定义的变量，渲染页面就会失败，出现类似如下的错误：

```
...
File "/home/ubuntu/r/venv/lib/python2.7/site-packages/mako/runtime.py", line 892, in
    _exec_template
    callable_(context, *args, **kwargs)
File "activity_index_html", line 224, in render_body
File "_activity_widgets_widget_html", line 1373, in render_widget

File "_activity_widgets_filmawards_html", line 117, in render_main

File "/home/ubuntu/r/venv/lib/python2.7/site-packages/mako/filters.py", line 78, in
    decode
    return decode(str(x))
File "/home/ubuntu/r/venv/lib/python2.7/site-packages/mako/runtime.py", line 226, in
    __str__
    raise NameError("Undefined")
NameError: Undefined
```

这个例子跨了多个模板文件，`_activity_widgets_widget_html`、`_activity_widgets_filmawards_html` 这样的模板并不是真实存在的模板文件，它们都是由 Mako 编译而成的临时结果，即便使用 `werkzeug.debug.DebuggedApplication` 也无法定位报错的行数。

这个时候可以找到 Mako 的源码中的 `template.py`（<https://github.com/zzzseek/mako/blob/master/mako/template.py>），在 `_compile`（<https://github.com/zzzseek/mako/blob/master/mako/template.py#L651>）这个函数结尾处添加两行打印输出：

```
def _compile(template, text, filename, generate_magic_comment):
    ...

    for index, s in enumerate(source.splitlines()):
        print index, s

    return source, lexer
```

这样，在终端就可以看到编译后的模板的内容，从而定位错误原因了。

使用 MySQL

数据是动态网站的基础，如果把数据放进数据库，就可以通过数据库管理系统对数据进行管理。MySQL 是一个开源的关系型数据库管理系统。它性能高、免费、配置简单、可靠性好，已经成为最流行的开源数据库。

在 Flask 应用中可以自由使用 MySQL、PostgreSQL、SQLite、Redis、MongoDB 来写原生的语句实现功能，也可以使用更高级别的数据库抽象方式，如 `SQLAlchemy` 或 `MongoEngine`

这样的 OR (D) M。本节将演示用 MySQL 的 Python 驱动 (MySQLdb) 写原生语句, 通过 SQLAlchemy 演示 ORM 的使用, 让大家熟悉 Web 开发中的数据库开发工作。

安装 MySQL 和驱动

首先安装 MySQL:

```
> sudo apt-get install mysql-server libmysqlclient-dev -yq
> sudo /etc/init.d/mysql start
```

安装之后默认已经启动了 MySQL。安装 mysql-server 的过程中需要指定 root 账户的密码, 生产环境一定要设置强复杂度的密码。

现在安装 MySQLdb:

```
> pip install mysql-python
```



安装用的名字是 mysql-python, 而不是 MySQLdb。

设置应用账号和权限

root 的权限很大, 不应该在 Web 应用中直接使用此用户 (root 密码一般都会被写进应用的配置文件中), 应该使用一个单独的用户。我们将执行如下三步:

1. 创建一个数据库 r。
2. 创建一个用户, 名为 web, 密码为 web。
3. 设置用户权限。用户 web 对数据库 r 有全部权限。

下面是执行过程:

```
> sudo mysql -u root
mysql> create database r;
Query OK, 1 row affected (0.00 sec)

mysql> create user 'web'@'localhost' identified by 'web';
Query OK, 0 rows affected (0.00 sec)

mysql> use r;
Database changed
mysql> grant all on r.* TO 'web'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> quit;  
Bye
```

用 MySQLdb 写原生语句

下面会编写一系列数据库开发的例子，为了重复利用，把常量放到独立的 `consts.py` 文件里面：

```
HOSTNAME = 'localhost'  
DATABASE = 'r'  
USERNAME = 'web'  
PASSWORD = 'web'  
DB_URI = 'mysql://{}:{ }@{ }/{ }'.format(  
    USERNAME, PASSWORD, HOSTNAME, DATABASE)
```

我们看第一个例子（`example1.py`）：

```
1 import MySQLdb  
2 from consts import HOSTNAME, DATABASE, USERNAME, PASSWORD  
3  
4 try:  
5     con = MySQLdb.connect(HOSTNAME, USERNAME, PASSWORD, DATABASE)  
6     cur = con.cursor()  
7     cur.execute("SELECT VERSION()")  
8     ver = cur.fetchone()  
9     print "Database version : %s " % ver  
10 except MySQLdb.Error as e:  
11     print "Error %d: %s" % (e.args[0], e.args[1])  
12     exit(1)  
13 finally:  
14     if con:  
15         con.close()
```

分析一下这个程序的细节：

- 第 5 行，`MySQLdb.connect` 返回的 `con` 是一个数据库连接实例。使用完通过 `con` 关闭数据库连接是一个好习惯。
- 第 6 行，`con.cursor()` 返回一个游标，数据库操作都是在游标实例上执行的。
- 第 7 行，`cur.execute` 方法传入的就是要执行的 SQL 语句。
- 第 8 行，`cur.fetchone()` 返回执行结果，这一点一开始可能不适应，因为数据库操作的结果不是在 `execute` 中直接返回的，而需要使用 `fetchone`、`fetchall`、`fetchmany` 这样的方法获取结果。

数据库操作主要以 CRUD 为主。CRUD 是 Create（创建）、Read（读取）、Update（更新）和 Delete（删除）的缩写。再来看一个 CRUD 的例子（curd.py）：

```
import MySQLdb
from consts import HOSTNAME, DATABASE, USERNAME, PASSWORD

con = MySQLdb.connect(HOSTNAME, USERNAME, PASSWORD, DATABASE)

with con as cur:
    cur.execute('drop table if exists users')
    cur.execute('create table users(Id INT PRIMARY KEY AUTO_INCREMENT, '
                'Name VARCHAR(25))')
    cur.execute("insert into users(Name) values('xiaoming')")
    cur.execute("insert into users(Name) values('wanglang')")
    cur.execute('select * from users')

    rows = cur.fetchall()
    for row in rows:
        print row
    cur.execute('update users set Name=%s where Id=%s', ('ming', 1))
    print 'Number of rows updated:', cur.rowcount

    cur = con.cursor(MySQLdb.cursors.DictCursor)
    cur.execute('select * from users')

    rows = cur.fetchall()
    for row in rows:
        print row['Id'], row['Name']
```

这次使用了 with 语句。connect 的 __enter__ 方法返回了游标，在 with 中执行结束，它会判断当前是否有错误，有错误就回滚，没有则进行事务提交，相当于无须自己来写下面这样的异常处理：

```
try:
    cur = con.cursor()
    cur.execute("insert into example(Name) values('xiaoming')")
    con.commit()
except MySQLdb.Error as e:
    con.rollback()
```

事务提交和回滚

事务主要用于处理操作量大、复杂度高的数据。如果操作的是一系列的动作，比如删除一个用户，不仅需要删除用户表中对应的记录，也要删除和用户表关联的表中对应的数据，

甚至还有其它业务上的需要。这个时候事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。



MySQL 的 InnoDB 引擎支持事务，而默认的 MyISAM 引擎不支持，需要根据业务来取舍。

ORM 简介

随着项目越来越大，采用写原生 SQL 的方式在代码中会出现大量的 SQL 语句，那么问题就出现了：

1. SQL 语句重复利用率不高，越复杂的 SQL 语句条件越多，代码越长。你会看到很多很相近的 SQL 语句。
2. 很多 SQL 语句是在业务逻辑中拼出来的，如果有数据库需要更改，就要求开发人员非常了解这些逻辑，否则就很容易漏掉对某些 SQL 语句的修改。
3. 写 SQL 时容易忽略 Web 安全问题，给未来造成隐患。

ORM，全称 Object Relational Mapping，中文叫作对象关系映射，通过它我们可以直接使用 Python 的类的方式做数据库开发，而不再直接写原生的 SQL 语句（甚至不需要 SQL 的基础）。通过把表映射成类，把行作为实例，把字段作为属性，ORM 在执行对象操作的时候会把对应的操作转换为数据库原生语句的方式来完成数据库开发工作。

ORM 有如下优点：

- 易用性。使用这种 ORM 数据库抽象封装方式做开发，可以有效减少出现重复 SQL 语句的概率，写出来的模型也更直观、清晰。
- 性能损耗小。ORM 转换成底层数据库操作指令确实会有一些开销。以笔者的经验，性能损耗很少（不足 5%），只要不是对性能有严苛的要求，综合考虑开发效率的提升、代码重复利用率等因素，带来的好处要远大于性能损耗，而且项目越大其作用越明显。
- 设计灵活。可以很轻松地写复杂的查询。
- 可移植。比如 SQLAlchemy，它支持多个关系数据库引擎，包括流行的 MySQL、PostgreSQL 和 SQLite。可以近乎无痛地换数据库，只需要改很少的配置项即可。

使用 SQLAlchemy

SQLAlchemy 是最流行的关系型数据库的 ORM 框架，它由 Mako 的作者 Mike Bayer 创建。我们先安装它：

```
> pip install SQLAlchemy
```

连接数据库

首先需要连接到数据库：

```
In : from sqlalchemy import create_engine
In : engine = create_engine('sqlite://', echo=False)
In : with engine.connect() as con:
...:     rs = con.execute('SELECT 1')
...:     print rs.fetchone()
...:
(1,)
```

`create_engine` 传入了一个数据库的 URI，`sqlite://` 表示使用了一个 SQLite 的内存型数据库。URI 的格式如下：

```
dialect+driver://username:password@host:port/database
```

`dialect` 是数据库的实现，比如 MySQL、PostgreSQL、SQLite。`driver` 是 Python 对应的驱动，如果不指定，会选择默认的驱动。比如 MySQL 的默认驱动是 `MySQLdb`：

```
engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
```

因为 `mysqldb` 是默认的驱动，所以可以不写驱动部分：

```
engine = create_engine('mysql://scott:tiger@localhost/foo')
```

执行 SQL 时也可以更简略地这样写：

```
In : rs = engine.execute('select 1')
In : rs.fetchone()
Out: (1,)
```



如果你需要详细的输出，可以设置 `echo=True`。

使用原生 SQL

我们把之前的例子 `curd.py` 改写成使用 SQLAlchemy 的 (`raw_sql.py`):

```
from sqlalchemy import create_engine
from consts import DB_URI

eng = create_engine(DB_URI)
with eng.connect() as con:
    con.execute('drop table if exists users')
    con.execute('create table users(Id INT PRIMARY KEY AUTO_INCREMENT, '
                'Name VARCHAR(25))')
    con.execute("insert into users(name) values('xiaoming')")
    con.execute("insert into users(name) values('wanglang')")
    rs = con.execute('select * from users')
    for row in rs:
        print row
```

它和之前的 MySQLdb 例子的不同之处在于，结果通过返回值获取，不再需要执行 `fetchone` 或者 `fetchall` 也能获取到。

使用表达式

SQLAlchemy 支持使用表达式的方式来操作数据库，这种方式 and Ruby On Rails 中的 Active Record 模式很像 (`exp_sql.py`):

```
from sqlalchemy import (create_engine, Table, MetaData, Column, Integer,
                        String, tuple_)
from sqlalchemy.sql import select, asc, and_
from consts import DB_URI

eng = create_engine(DB_URI)

meta = MetaData(eng)
users = Table(
    'Users', meta,
    Column('Id', Integer, primary_key=True, autoincrement=True),
    Column('Name', String(50), nullable=False),
)

if users.exists():
    users.drop()

def execute(s):
```

```
print '-' * 20
rs = con.execute(s)
for row in rs:
    print row['Id'], row['Name']

with eng.connect() as con:
    for username in ('xiaoming', 'wanglang', 'lilei'):
        user = users.insert().values(Name=username)
        con.execute(user)

stm = select([users]).limit(1)
execute(stm)

k = [(2,)]
stm = select([users]).where(tuple_(users.c.Id).in_(k))
execute(stm)

stm = select([users]).where(and_(users.c.Id > 2,
                                users.c.Id < 4))
execute(stm)

stm = select([users]).order_by(asc(users.c.Name))
execute(stm)

stm = select([users]).where(users.c.Name.like('%min%'))
execute(stm)
```

分析一下这个例子：

- 使用 `users.create()` 的方式来创建表，如果需要创建的表比较多，也可以选择使用 `meta.create_all(eng)`。
- 代码中的 `stm` 变量就是一个生成好的 SQL 语句。比如“`stm = select([users]).order_by(asc(users.c.Name))`”，`stm` 的结果是：

```
SELECT `Users`.`Id`, `Users`.`Name`
FROM `Users` ORDER BY `Users`.`Name` ASC
```

使用 ORM

ORM 是基于 SQLAlchemy 表达式语言的，看一个例子（`orm_sql.py`）：

```
from sqlalchemy import create_engine, Column, Integer, String, Sequence
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import and_, or_
```

```
from sqlalchemy.orm import sessionmaker

from consts import DB_URI

eng = create_engine(DB_URI)
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, Sequence('user_id_seq'),
                 primary_key=True, autoincrement=True)
    name = Column(String(50))

Base.metadata.drop_all(bind=eng) # 删除表
Base.metadata.create_all(bind=eng)

Session = sessionmaker(bind=eng)
session = Session()

session.add_all([User(name=username)
                 for username in ('xiaoming', 'wanglang', 'lilei')])

session.commit()

def get_result(rs):
    print '-' * 20
    for user in rs:
        print user.name

rs = session.query(User).all()
get_result(rs)
rs = session.query(User).filter(User.id.in_([2, ]))
get_result(rs)
rs = session.query(User).filter(and_(User.id > 2, User.id < 4))
get_result(rs)
rs = session.query(User).filter(or_(User.id == 2, User.id == 4))
get_result(rs)
rs = session.query(User).filter(User.name.like('%min%'))
get_result(rs)
user = session.query(User).filter_by(name='xiaoming').first()
get_result([user])
```

分析一下这个例子：

- 定义的 User 类会生成一张表，__tablename__ 的值就是表名。
- 通过 sessionmaker 创建一个会话，会话提供了事务控制的支持。模型实例对象本身独立存在，如果要想让其修改（创建）生效，需要把它们加入某个会话；如果不希望对其生效就从会话中去掉由 session 管理的实例对象。执行 session.commit() 时修改被提交到数据库，执行 session.rollback() 可以回滚变更。



例子里在执行开始前会先删除表，但是实际工作请不要这样用。

还可以通过 sqlalchemy.text 写复杂的条件语句来操作数据库（text_sql.py）：

```
rs = session.query(User).filter(
    text('id > 2 and id < 4')).order_by(text('id')).all()
get_result(rs)
rs = session.query(User).filter(text('id<:value and name=:name')).params(
    value=3, name='xiaoming').all()
get_result(rs)
rs = session.query(User).from_statement(
    text('SELECT * FROM users where name=:name')).params(name='wanglang').all()
get_result(rs)
```

数据库关联

InnoDB 类型的表可以使用外键进行多表关联，保证数据的一致性和实现一些级联操作。我们看一个例子（rel_sql.py）：

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship

from consts import DB_URI

eng = create_engine(DB_URI)
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(50))
```

```

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True, autoincrement=True)
    email_address = Column(String(128), nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'))
    # user 字段关联到 User 表
    user = relationship('User', back_populates='addresses')

# 这个字段需要放在 Address 类定义之后
User.addresses = relationship('Address', order_by=Address.id,
                              back_populates='user')

Base.metadata.drop_all(bind=eng)
Base.metadata.create_all(bind=eng)

Session = sessionmaker(bind=eng)
session = Session()

user = User(name='xiaoming')
user.addresses = [Address(email_address='a@gmail.com', user_id=user.id),
                  Address(email_address='b@gmail.com', user_id=user.id)]
session.add(user)
session.commit()

```

Address 表的 user_id 字段其实就是 User 的 id 字段，使用 ForeignKey 关联之后，就不需要在 Address 上独立存储一份 user_id 数据。我们通过 “ipython -i” 执行上面的脚本并继续验证：

```

> ipython --no-banner -i chapter3/section3/rel_sql.py
In : for u, a in session.query(User, Address).\
...: filter(User.id==Address.user_id).\
...: filter(Address.email_address=='b@gmail.com').\
...: all():
...:     print 'User ID: {}'.format(u.id)
...:     print 'Email Address: {}'.format(a.email_address)
...:
User ID: 1
Email Address: b@gmail.com
In : session.query(Address).join(User).filter(Address.id.in_([2])).all()[0].
    email_address
Out: 'b@gmail.com'
In : session.query(User).join(Address).filter(Address.email_address=='a@gmail.com').one
    ().name
Out: 'xiaoming'

```




虽然使用外键可以降低开发成本，减少数据量，但是在用户量大、并发度高的时候，不推荐使用外键来关联，数据的一致性和完整性问题可以通过事务来保证。

在 Flask 中使用 SQLAlchemy

Flask-SQLAlchemy (<https://github.com/mitsuhiko/flask-sqlalchemy>) 可以帮助我们在 Flask 中很方便地使用 SQLAlchemy，它是 Flask 作者写的扩展。首先安装 Flask-SQLAlchemy：

```
> pip install Flask-SQLAlchemy
```

一个大型项目，模型应该对应地存放在不同的模型文件中，我们把 `users` 这个表的模型存放到 `users.py`：

```
from ext import db

class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50))

    def __init__(self, name):
        self.name = name
```

`db.Model` 其实还是基于 `declarative_base` 实现的，Flask-SQLAlchemy 提供了一个和 Django 风格很像的基类。在这里重新定义了 `User` 的 `__init__` 方法，因为它默认需要传入所有字段，而 `id` 是一个自增长的字段，不需要传入。

`ext` 模块存放了 Flask 第三方的扩展：

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

这样的好处是，`db` 是一个没有依赖的常量，`app` 也可以 “`from ext import db`”，不会造成循环依赖。

把配置也独立出来，放入 `config.py` 中：

```
from consts import DB_URI
DEBUG = True
```

```
SQLALCHEMY_DATABASE_URI = DB_URI
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

现在，看一个能在 Web 上创建用户的应用（`app_with_sqlalchemy.py`）：

```
1 from flask import Flask, request, jsonify
2
3 from ext import db
4 from users import User
5
6 app = Flask(__name__)
7 app.config.from_object('config')
8 db.init_app(app)
9
10 with app.app_context():
11     db.drop_all()
12     db.create_all()
13
14
15 @app.route('/users', methods=['POST'])
16 def users():
17     username = request.form.get('name')
18
19     user = User(username)
20     print 'User ID: {}'.format(user.id)
21     db.session.add(user)
22     db.session.commit()
23
24     return jsonify({'id': user.id})
25
26
27 if __name__ == '__main__':
28     app.run(host='0.0.0.0', port=9000)
```

上述例子有如下细节：

- 第 7 行，使用 `from_object` 加载 `config.py` 里的配置，生产环境中推荐这样管理配置。
- 第 8 行，把第三方扩展放在 `ext.py` 之后，这里只需要使用 `xx.init_app(app)` 的方式初始化，这也是推荐的用法。
- 第 10-12 行，`drop_all` 和 `create_all` 要在定义 `model` 之后再执行。Flask-SQLAlchemy 要求执行的时候有应用上下文，但是在这里还没有，所以需要使用“`with app.app_context()`”创建应用上下文。关于应用上下文稍后会详细介绍，也会介绍正确的用法。
- 第 20 行，`print` 的 `user.id` 永远是 `None`，因为在没有 `commit` 之前还没有创建它。`commit` 之后 `user.id` 会自动改成在表中创建的条目 `id`。

记录慢查询

数据库性能是开发者必须关注的重点之一，在复杂的业务代码逻辑前提下，如果只是通过 MySQL 的日志去看慢查询的日志很难定位问题，那么可以借用 `SQLALCHEMY_RECORD_QUERIES` 和 `DATABASE_QUERY_TIMEOUT` 将慢查询及相关上下文信息记录到日志中。下面展示一个基于 `app_with_sqlalchemy.py` 的例子（`logger_slow_query.py`）：

```
import logging
from logging.handlers import RotatingFileHandler

from flask_sqlalchemy import get_debug_queries

...
app.config['DATABASE_QUERY_TIMEOUT'] = 0.0001
app.config['SQLALCHEMY_RECORD_QUERIES'] = True

formatter = logging.Formatter(
    "[%(asctime)s] {%(pathname)s:%(lineno)d} %(levelname)s - %(message)s")
handler = RotatingFileHandler('slow_query.log', maxBytes=10000, backupCount=10)
handler.setLevel(logging.WARN)
handler.setFormatter(formatter)
app.logger.addHandler(handler)

@app.after_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= app.config['DATABASE_QUERY_TIMEOUT']:
            app.logger.warn(
                ('Context:{{}\nSLOW QUERY: {{}\nParameters: {{}\n'
                 'Duration: {{}\n').format(query.context, query.statement,
                                           query.parameters, query.duration))
    return response
```

这个例子主要做了 3 件事：

1. 启用查询记录功能。
2. 给 `app.logger` 添加一个记录日志到名为 `slow_query.log` 的文件的处理器，这个日志会按大小切分。
3. 添加 `after_request` 钩子，每次请求结束后获取执行的查询语句，假如超过阈值则记录日志。

我们看一下效果：

```
➤ http -f post http://localhost:9000/users name=xiaoming
```

执行完成后，除了在启动 Flask 应用的终端之外，还会在 `slow_query.log` 日志中看到如下记录：

```
[2016-07-14 20:50:11,145] {chapter3/section3/logger_slow_query.py:48} WARNING -  
Context:chapter3/section3/logger_slow_query.py:36 (users)  
SLOW QUERY: INSERT INTO users (name) VALUES (%s)  
Parameters: ('lihang',)  
Duration: 0.00047492980957
```

日志中包含了出现问题的代码位置以及对应的 SQL 语言，我们就直接知道问题的根源了。



上例中的变量 `DATABASE_QUERY_TIMEOUT` 的值为 0.0001 只是为了演示，生产环境需要按需调大这个阈值。

理解 Context

Context（上下文）是 Flask 里面非常好的设计，使用 Flask 需要非常理解应用上下文和请求上下文这两个概念。在开始正文之前我们先了解一些必要的知识。

本地线程

本地线程（Thread Local）希望不同的线程对于内容的修改只在线程内发挥作用，线程之间互不影响（`local.py`）：

```
import threading  
mydata = threading.local()  
mydata.number = 42  
print mydata.number  
log = []
```

```
def f():  
    mydata.number = 11  
    log.append(mydata.number)
```

```
thread = threading.Thread(target=f)  
thread.start()
```

```
thread.join()
print log
print mydata.number
> python chapter3/local.py
42
[11] # 在线程内变成了mydata.number其他的值
42 # 但是没有影响到开始设置的值
```

本地线程实现的原理就是：在 `threading.current_thread().__dict__` 里添加一个包含对象 `mydata` 的 `id` 值的 `key`，来保存不同线程的状态。

Werkzeug 的 Local

Werkzeug 自己实现了本地线程。`werkzeug.local.Local` 和 `threading.local` 的区别如下：

- Werkzeug 使用了自定义的 `__storage__` 保存不同线程下的状态。
- Werkzeug 提供了释放本地线程的 `release_local` 方法。
- Werkzeug 使用如下方法得到 `get_ident` 函数，用来获得线程/协程标识符。

```
try:
    from greenlet import getcurrent as get_ident
except ImportError:
    try:
        from thread import get_ident
    except ImportError:
        from _thread import get_ident
```

假如已经安装了 `Greenlet`，会优先选择 `Greenlet`，否则使用系统线程。`Greenlet` 是以 C 扩展模块形式接入 Python 的轻量级协程，它运行在操作系统进程的内部，但是会被协作式地调度。

Werkzeug 还实现了两种数据结构。

- `LocalStack`：基于 `werkzeug.local.Local` 实现的栈结构，可以将对象推入、弹出，也可以快速拿到栈顶对象。
- `LocalProxy`：作用和名字一样，是标准的代理模式。构造此结构时接受一个可以调用的参数（一般是函数），这个函数执行后就是通过 `LocalStack` 实例化的栈的栈顶对象。对于 `LocalProxy` 对象的操作实际上都会转发到这个栈顶对象（也就是一个 `Thread Local` 对象）上面。

flask.request

先看一个极简的示例：

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/people/')
def people():
    name = request.args.get('name')
```

仔细想一下，这里先引用了 `flask.request`，但是直到用户访问 `/people/` 的时候才通过 `request.args.get('name')` 获得请求的参数值。试想，引用的时候还没有发生这个请求，那么请求上下文是怎么获得的呢？

`flask.request` 就是一个获取名为 `_request_ctx_stack` 的栈顶对象的 `LocalProxy` 实例：

```
from functools import partial
from werkzeug.local import LocalProxy

def _lookup_req_object(name):
    top = _request_ctx_stack.top
    if top is None: # 所以注意，一定要先把请求推入堆栈再调用
        raise RuntimeError('working outside of request context')
    return getattr(top, name)

request = LocalProxy(partial(_lookup_req_object, 'request'))
```

上述逻辑能正常使用，是因为其流程是这样的：

- 用户访问产生请求。
- 在发生请求的过程中向 `_request_ctx_stack` 推入这个请求上下文的对象，它会变成栈顶。`request` 就会成为这个请求上下文，也就包含了这次请求相关的信息和数据。
- 在视图函数中使用 `request` 就可以使用 `request.args.get('name')` 了。

设想不使用 `LocalStack` 和 `LocalProxy` 的话，要想让视图函数访问到请求对象，就只能将其作为参数，一步步传入视图函数中。这样做的缺点是会让每个视图函数都增加一个 `request` 参数，而 `Flask` 巧妙地使用上下文把某些对象变为全局可访问（实际上是特定环境的局部对象的代理），每个线程看到的上下文对象却是不同的，这样就巧妙地解决了这个问题。

使用上下文

应用上下文的典型应用场景是缓存一些在发生请求之前要使用到的资源，比如生成数据库连接和缓存一些对象；请求上下文发生在 HTTP 请求开始，WSGI Server 调用 Flask.__call__() 之后。应用上下文并不是应用启动之后生成的唯一上下文，我们看一下它们的关系：

```
class RequestContext(object):
    self._implicit_app_ctx_stack = []

    def push(self):
        # some stuff
        app_ctx = _app_ctx_stack.top
        if app_ctx is None or app_ctx.app != self.app:
            app_ctx = self.app.app_context()
            app_ctx.push()
            self._implicit_app_ctx_stack.append(app_ctx)
        # some other stuff
```

也就是说应用上下文是被动的在推入请求上下文的过程中生成的，在请求结束的时候，也会把请求上下文弹出：

```
class RequestContext(object):
    def pop(self, exc=_sentinel):
        app_ctx = self._implicit_app_ctx_stack.pop()
        try:
            # some stuff
        finally:
            # some other stuff
            if app_ctx is not None:
                app_ctx.pop(exc)
```

也就是说，事实上在 Web 应用环境中，请求上下文和应用文是一一对应的。请求上下文和应用上下文都是本地线程的，那么区分它们有什么意义呢？

- 使用本章稍后介绍的中间件 DispatcherMiddleware，支持多个 app 共存。就像 request 一样，在多 app 情况下之前也要保证 app 之间的隔离。
- 非 Web 模式下。比如进行测试，一个应用上下文可以有多个请求上下文。但是不能执行 pop 方法，或者使用 with 语句（__exit__ 中会自动执行 pop 方法）。

Flask 中有 4 个上下文变量。

1. flask.current_app：应用上下文。它是当前 app 实例对象。
2. flask.g：应用上下文。处理请求时用作临时存储的对象。

3. flask.request: 请求上下文。它封装了客户端发出的 HTTP 请求中的内容。
4. flask.session: 请求上下文。它存储了用户会话。

其中最常见的是 flask.g 和 flask.request，在前面已经使用过 flask.request，现在对 3.3 节的 app_with_sqlalchemy.py 扩充，添加上下文的钩子等功能（app.py）：

```
import random

from flask import Flask, g, render_template
from ext import db
from users import User
app = Flask(__name__, template_folder='.././templates')
app.config.from_object('config')
db.init_app(app)

def get_current_user():
    users = User.query.all()
    return random.choice(users)

@app.before_first_request
def setup():
    db.drop_all()
    db.create_all()
    fake_users = [
        User('xiaoming', 'xiaoming@dongwm.com'),
        User('dongwweiming', 'dongwm@dongwm.com'),
        User('admin', 'admin@dongwm.com')
    ]
    db.session.add_all(fake_users)
    db.session.commit()

@app.before_request
def before_request():
    g.user = get_current_user()

@app.teardown_appcontext
def teardown(exc=None):
    if exc is None:
        db.session.commit()
    else:
        db.session.rollback()
    db.session.remove()
```



```
g.user = None

@app.context_processor
def template_extras():
    return {'enumerate': enumerate, 'current_user': g.user}

@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404

@app.template_filter('capitalize')
def reverse_filter(s):
    return s.capitalize()

@app.route('/users')
def user_view():
    users = User.query.all()
    return render_template('chapter3/section4/user.html', users=users)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

例子中有 6 个钩子装饰器，被装饰的函数会注册到 `app` 中，它们将在不同的阶段执行。

- `before_first_request`: 在处理第一次请求之前执行。
- `before_request`: 在每次请求前执行。
- `teardown_appcontext`: 不管是否有异常，注册的函数都会在每次请求之后执行。
- `context_processor`: 上下文处理的装饰器，返回的字典中的键可以在上下文中使用。
- `template_filter`: 在使用 Jinja2 模板的时候可以方便地注册过滤器。
- `errorhandler`: `errorhandler` 接收状态码，可以自定义返回这种状态码的响应的处理方法。

我们先详细地分析一下：

- `setup` 函数常用来初始化数据，尤其是开发环境下，每次启动应用都会先删掉之前创建的假数据再重新创建。
- 之前说 `flask.g` 是一个应用上下文，通常放在 `before_request` 中对它进行数据的填充。

- 一般来说，对资源的操作有一个 `get_X` 和一个 `teardown_X` 对应，多个资源的使用可以使用同一个 `teardown` 函数。`teardown` 通常是做一些环境的清理工作，提交未提交的操作请求等，在本地开发环境和测试时意义较大。
- 由于 Jinja2 模板的限制，并不能直接使用 `enumerate` 这样的 Python 自带的函数（虽然 Jinja2 支持在 `for` 循环中使用 `loop.index` 和 `loop.index0`，但是无法满足全部需要），可以使用 `context_processor` 把要用到的上下文资源传进去。这样在模板中就可以直接使用 `enumerate` 和 `current_user` 了。
- `errorhandler` 除了可自定义对不同错误状态码的返回内容，还可以传入自定义的异常对象。
- 虽然 Jinja2 支持了非常多的过滤器，但还是无法满足我们的全部需要。注册一个新的过滤器很方便，这个例子中注册了一个叫作 `capitalize` 的过滤器，在模板中可以这样使用 “`{{ user.name | capitalize }}`”。

使用 LocalProxy 替代 g

`g` 也是一个被 `LocalProxy` 包装的对象，而且还需要借助 `before_request` 这个钩子，可不可以更简化呢？现在我们对 `app.py` 稍加修改（`app_with_local_proxy.py`），创建一个全局可访问的 `current_user` 来直接使用：

```
from werkzeug.local import LocalProxy
current_user = LocalProxy(get_current_user)
```

现在就可以去掉 `before_request`，直接使用 `current_user` 了：

```
@app.context_processor
def template_extras():
    return {'enumerate': enumerate, 'current_user': current_user}
```

从零开始实现一个文件托管服务

基于之前所讲的知识，本节我们来实现一个真实的应用。这是一个文件托管服务，主要解决以下问题：

- 上传后的文件可以被永久存放。
- 上传后的文件有一个功能完备的预览页。预览页显示文件大小、文件类型、上传时间、下载地址和短链接等信息。
- 可以通过传参数对图片进行缩放和剪切。

- 不错的页面展示效果。
- 为节省空间，相同文件不重复上传，如果文件已经上传过，则直接返回之前上传的文件。

我们先安装一些之前没有安装的依赖：

```
> sudo apt-get install libjpeg8-dev -yq
> pip install -r chapter3/section5/requirements.txt
```

requirements.txt 中包含以下内容。

- python-magic: libmagic 的 Python 绑定，用于确定文件类型。
- Pillow: PIL (Python Imaging Library) 的分支，用来替代 PIL。
- cropresize2: 用来剪切和调整图片大小。
- short_url: 创建短链接。

文件托管服务的建表语句如下 (databases/schema.sql)：

```
CREATE TABLE `PasteFile` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `filename` varchar(5000) NOT NULL,
  `filehash` varchar(128) NOT NULL,
  `filemd5` varchar(128) NOT NULL,
  `uploadtime` datetime NOT NULL,
  `mimetype` varchar(256) NOT NULL,
  `size` int(11) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `filehash` (`filehash`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

其中指定了“ENGINE=InnoDB”，这个表会使用 InnoDB 引擎。直接在命令行把它导入到数据库：

```
> (echo "use r"; cat databases/schema.sql) | mysql --user='web' --password='web'
```

这个项目有多个文件。

1. config.py: 用于存放配置。

```
SQLALCHEMY_DATABASE_URI = 'mysql://web:web@localhost:3306/r'
UPLOAD_FOLDER = '/tmp/permdir'
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

UPLOAD_FOLDER 指定了存放上传文件的目录。我们先创建这个目录：

```
> mkdir /tmp/permdir
```

2. `utils.py`: 用于存放功能函数。

- `get_file_md5`: 获得文件的 md5 值。
- `humanize_bytes`: 返回可读的文件大小。

```
In : humanize_bytes(100)
Out: '100.00 bytes'
In : humanize_bytes(34500)
Out: '33.00 kB'
In : humanize_bytes(34500000)
Out: '32.00 MB'
```

- `get_file_path`: 根据上传文件的目录获得文件路径。

3. `mimes.py`: 只接受文件中定义了的媒体类型。

```
AUDIO_MIMES = [
    'audio/ogg',
    'audio/mp3'
    ...
]
```

```
IMAGE_MIMES = [
    'image/jpeg',
    'image/png',
    ...
]
```

```
VIDEO_MIMES = [
    'video/mp4',
    'video/ogg',
    ...
]
```

4. `ext.py`: 存放扩展的封装。

```
from flask_mako import MakoTemplates, render_template
from flask_sqlalchemy import SQLAlchemy

mako = MakoTemplates()
db = SQLAlchemy()
```

5. `models.py`: 存放模型。

6. `app.py`: 应用主程序。

models.py 文件中只包含了 PasteFile 模型。它的字段定义和初始化方法如下：

```
from ext import db

class PasteFile(db.Model):
    __tablename__ = 'PasteFile'
    id = db.Column(db.Integer, primary_key=True)
    filename = db.Column(db.String(5000), nullable=False)
    filehash = db.Column(db.String(128), nullable=False, unique=True)
    filemd5 = db.Column(db.String(128), nullable=False, unique=True)
    uploadtime = db.Column(db.DateTime, nullable=False)
    mimetype = db.Column(db.String(256), nullable=False)
    size = db.Column(db.Integer, nullable=False)

    def __init__(self, filename='', mimetype='application/octet-stream',
                  size=0, filehash=None, filemd5=None):
        self.uploadtime = datetime.now()
        self.mimetype = mimetype
        self.size = int(size)
        self.filehash = filehash if filehash else self._hash_filename(filename)
        self.filename = filename if filename else self.filehash
        self.filemd5 = filemd5

    @staticmethod
    def _hash_filename(filename):
        _, _, suffix = filename.rpartition('.')
        return '%s.%s' % (uuid.uuid4().hex, suffix)
```

现在看一下 app 的初始化：

```
from werkzeug import SharedDataMiddleware

from ext import db, makro
from utils import get_file_path

app = Flask(__name__, template_folder='.././templates/r',
            static_folder='.././static')
app.config.from_object('config')

app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/i/': get_file_path()
})

makro.init_app(app)
db.init_app(app)
```

上述例子有如下细节需要注意：

- 使用 `SharedDataMiddleware` 是实现在页面读取源文件的最简单的方法。
- 只是把第三方扩展初始化放在了 `app.py` 中，而没有使用 “`db = SQLAlchemy(app)`” 这样的方式。这是因为在大型应用中如果 `db` 被多个模型文件引用的话，会造成 “`from app import db`” 这样的方式，但是往往也在 `app.py` 中也会引用模型文件定义的类，这就造成了循环引用。所以最好的做法是把它放在不依赖其他模块的独立文件中。

我们来分别看不同的视图及其实现逻辑。

首页

首页就是上传图片页，通过这个页面可以上传图片，并生成预览页：

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        uploaded_file = request.files['file']
        w = request.form.get('w')
        h = request.form.get('h')
        if not uploaded_file:
            return abort(400)

        if w and h:
            paste_file = PasteFile.rsize(uploaded_file, w, h)
        else:
            paste_file = PasteFile.create_by_upload_file(uploaded_file)
        db.session.add(paste_file)
        db.session.commit()

    return jsonify({
        'url_d': paste_file.url_d,
        'url_i': paste_file.url_i,
        'url_s': paste_file.url_s,
        'url_p': paste_file.url_p,
        'filename': paste_file.filename,
        'size': humanize_bytes(paste_file.size),
        'time': str(paste_file.uploadtime),
        'type': paste_file.type,
        'quoteurl': paste_file.quoteurl
    })
    return render_template('index.html', **locals())
```

如果是 GET 请求,直接渲染 index.html。如果是 POST 方法,通过 PasteFile.create_by_upload_file 创建一个 PasteFile 实例:

```
@classmethod
def get_by_md5(cls, filemd5):
    return cls.query.filter_by(filemd5=filemd5).first()

@property
def path(self):
    return get_file_path(self.filehash)

@classmethod
def create_by_upload_file(cls, uploaded_file):
    rst = cls(uploaded_file.filename, uploaded_file.mimetype, 0)
    uploaded_file.save(rst.path)
    with open(rst.path) as f:
        filemd5 = get_file_md5(f)
        uploaded_file = cls.get_by_md5(filemd5)
        if uploaded_file:
            os.remove(rst.path)
            return uploaded_file
    filestat = os.stat(rst.path)
    rst.size = filestat.st_size
    rst.filemd5 = filemd5
    return rst
```

创建 PasteFile 实例前会先保存文件,保存的文件名是 rst.path。如果通过被上传文件的 md5 值判断的文件之前已经上传过,则直接删掉这个文件,并返回之前创建的文件。

rst.path 使用了 filehash, filehash 是通过 _hash_filename 方法生成的随机名字,这是为了防止不同的用户上传的同名文件造成的替换。

如果上传请求是一个 POST 请求,并且指定了长和宽,会先裁剪图片再保存:

```
import cropresize2
from PIL import Image

@classmethod
def rsize(cls, old_paste, weight, height):
    assert old_paste.is_image, TypeError('Unsupported Image Type.')

    img = cropresize2.crop_resize(
        Image.open(old_paste.path), (int(weight), int(height)))

    rst = cls(old_paste.filename, old_paste.mimetype, 0)
    img.save(rst.path)
```

```

filestat = os.stat(rst.path)
rst.size = filestat.st_size
return rst

```

重新设置图片页

支持对现有的图片重新设置大小，返回新的图片地址：

```

@app.route('/r/<img_hash>')
def rsize(img_hash):
    w = request.args['w']
    h = request.args['h']

    old_paste = PasteFile.get_by_filehash(img_hash)
    new_paste = PasteFile.rsize(old_paste, w, h)

    return new_paste.url_i

```

其中 `get_by_filehash` 方法就是从数据库中找到匹配 `filehash` 的条目：

```
from flask import abort
```

```

@classmethod
def get_by_filehash(cls, filehash, code=404):
    return cls.query.filter_by(filehash=filehash).first() or abort(code)

```

`url_i` 属性获取的是源文件的地址。其他文件属性如下：

```

def get_url(self, subtype, is_symlink=False):
    hash_or_link = self.symlink if is_symlink else self.filehash
    return 'http://{host}/{subtype}/{hash_or_link}'.format(
        subtype=subtype, host=request.host, hash_or_link=hash_or_link)

@property
def url_i(self):
    return self.get_url('i')

@property
def url_p(self):
    return self.get_url('p')

@property
def url_s(self):
    return self.get_url('s', is_symlink=True)

```



```
@property
def url_d(self):
    return self.get_url('d')
```

通过 `get_url` 可以拼不同类型的请求地址，如表 3.1 所示。

表 3.1 不同类型的请求地址

方 法	作 用
<code>url_p</code>	文件预览地址
<code>url_d</code>	文件下载地址
<code>url_s</code>	文件短链接地址

下载页

下载文件时使用 “/d/img_hash.jpg” 这样的地址，可以用 Flask 提供的 `send_file` 实现：

```
from flask import send_file
```

```
ONE_MONTH = 60 * 60 * 24 * 30
```

```
@app.route('/d/<filehash>', methods=['GET'])
def download(filehash):
    paste_file = PasteFile.get_by_filehash(filehash)

    return send_file(open(paste_file.path, 'rb'),
                     mimetype='application/octet-stream',
                     cache_timeout=ONE_MONTH,
                     as_attachment=True,
                     attachment_filename=paste_file.filename.encode('utf-8'))
```

预览页

预览文件使用 “/p/img_hash.jpg” 这样的地址：

```
@app.route('/p/<filehash>')
def preview(filehash):
    paste_file = PasteFile.get_by_filehash(filehash)

    if not paste_file:
        filepath = get_file_path(filehash)
```

```
if not(os.path.exists(filepath) and (not os.path.islink(filepath))):
    return abort(404)

paste_file = PasteFile.create_by_old_paste(filehash)
db.session.add(paste_file)
db.session.commit()

return render_template('success.html', p=paste_file)
```

在首页上传完毕时也会在地址栏显示这样的地址，但事实上并没有发生跳转，只是用 JavaScript 修改了地址。由于它们使用了同一个文件卡片组件，所以看起来一模一样。

短链接页

由于文件 hash 值太长，支持使用短连接的方式访问，使用 “/s/short_url” 这样的地址：

```
@app.route('/s/<symlink>')
def s(symlink):
    paste_file = PasteFile.get_by_symlink(symlink)

    return redirect(paste_file.url_p)
```

但是并不需要把短链接存放进数据库，正确的做法是用 id 这个唯一标识生成短链接地址：

```
import short_url
from werkzeug.utils import cached_property
```

```
@cached_property
def symlink(self):
    return short_url.encode_url(self.id)
```

通过短链接获得对应数据库条目的方法如下：

```
@classmethod
def get_by_symlink(cls, symlink, code=404):
    id = short_url.decode_url(symlink)
    return cls.query.filter_by(id=id).first() or abort(code)
```

现在启动服务就可以看到效果了：

```
> mkdir /tmp/permdir
> python chapter3/section7/app.py
```

在线的效果可以访问搭建在 Heroku 上的 DEMO (<https://vast-brushlands-4477.herokuapp.com/>)。

第4章

Flask 开发进阶

本章将介绍 Flask 的一些高级用法，主要有如下内容：

- Flask 的信号机制。利用信号可以实现一部分的业务解耦。
- Flask 的一些常用、主流的扩展，如 Flask-Script、Flask-DebugToolbar、Flask-Migrate、Flask-WTF、Flask-Security、Flask-RESTful、Flask-Admin 和 Flask-Assets。每个扩展都至少包含一个完整的真实例子。
- Flask 的依赖库 Werkzeug 的使用。

Flask 的信号机制

项目功能越复杂，代码量越大，就越需要做业务解耦。否则在其之上做开发和维护是很痛苦的，尤其是对于团队的新人。Flask 从 0.6 开始，通过 Blinker (<http://bit.ly/28RwFDC>) 提供了信号支持。信号就是在框架核心功能或者一些 Flask 扩展发生动作时所发送的通知，用于帮助你解耦应用。

Blinker 的使用

Blinker 目前还不是 Flask 的默认依赖，所以信号是不可用的。我们需要先安装它：

```
> pip install blinker
```

先了解下 Blinker (blinker_signal.py):

```
from blinker import signal

started = signal('test-started')

def each(round):
    print 'Round {}'.format(round)

def round_two(round):
    print 'Only {}'.format(round)

started.connect(each)
started.connect(round_two, sender=2)

for round in range(1, 4):
    started.send(round)
```

其中 connect 是订阅信号的方法。第二个参数是可选的,用于确定信号的发送端。“started.connect(round_two, sender=2)” 表示值为 2 的时候才会接收。

send 是发送信号的方法。先看一下执行结果:

```
Round 1!
Round 2!
Only 2
Round 3!
```

可以设想, connect 和 send 这两个方法不放在一个文件中, 它们通过 started 作为桥梁达到解耦的作用。

之前我们介绍过 Flask 的钩子, 比如 before_request 和 after_request, 这些钩子不需要 Blinker 库并且允许你改变请求对象 (request) 或者响应对象 (response), 而信号和钩子做的事情很像, 只不过信号并不对请求对象和响应对象做改变, 仅承担记录 and 通知的工作。

Flask 中内置的信号

Flask 可以发送 9 种信号, 第三方的扩展中也可能会有额外的信号。而我们需要做的是添加对应的信号订阅。这里介绍常用的 6 种信号。

1. flask.template_rendered: 模板渲染成功的时候发送, 这个信号与模板实例 template、上下文的字典一起调用。

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)
```

```
from flask import template_rendered
template_rendered.connect(log_template_renders, app)
```

2. flask.request_started: 建立请求上下文后, 在请求处理开始前发送, 订阅者可以用 request 之类的标准全局代理访问请求。

```
def log_request(sender, **extra):
    sender.logger.debug('Request context is set up')
```

```
from flask import request_started
request_started.connect(log_request, app)
```

3. flask.request_finished: 在响应发送给客户端之前发送, 可以传递 response。

```
def log_response(sender, response, **extra):
    sender.logger.debug('Request context is about to close down. '
                        'Response: %s', response)
```

```
from flask import request_finished
request_finished.connect(log_response, app)
```

4. flask.got_request_exception: 在请求处理中抛出异常时发送, 异常本身会通过 exception 传递到订阅函数。

```
def log_exception(sender, exception, **extra):
    sender.logger.debug('Got exception during processing: %s', exception)
```

```
from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

5. flask.request_tearing_down: 在请求销毁时发送, 它总是被调用, 即使发生异常。

```
def close_db_connection(sender, **extra):
    session.close()
```

```
from flask import request_tearing_down
request_tearing_down.connect(close_db_connection, app)
```

6. flask.appcontext_tearing_down: 在应用上下文销毁时发送, 它总是被调用, 即使发生异常。

```
def close_db_connection(sender, **extra):
    session.close()
```

```
from flask import request_tearing_down
appcontext_tearing_down.connect(close_db_connection, app)
```

自定义信号

在自己的应用中可以直接使用 **Blinker** 创建信号，现在定义一种对于上传大文件的信号：

```
from blinker import Namespace
web_signals = Namespace()
large_file_saved = web_signals.signal('large-file-saved')
```

当上传文件大于一个阈值的时候，就可以发送这个信号。



当编写一个 Flask 扩展并且想优雅地在未安装 **Blinker** 时退出，可以使用 `flask.signals.Namespace`——在订阅信号的时候，如果发现未安装 **Blinker** 会抛出 `RuntimeError`。

信号订阅的高级用法

从 **Blinker** 1.1 开始可以用新的 `connect_via()` 装饰器订阅信号。比如使用上面介绍的 `flask.appcontext_tearing_down`：

```
@appcontext_tearing_down.connect_via(app)
def close_db_connection(sender, **extra):
    session.close()
```

也可以通过装饰器来用 `connect` 方法，比如下面的方法：

```
def each(round):
    print 'Round {}'.format(round)
```

```
started.connect(each)
```

可以简写成：

```
@started.connect
def each(round):
    print 'Round {}'.format(round)
```

Flask-Login 中的信号

Flask-Login 插件中带了 6 种信号 (<http://bit.ly/28PSm5F>)，可以基于其中的信号做一些额外工作，比如基于 `user_logged_in` 来记录用户的登录次数和登录 IP 等。

我们先安装它：

```
> pip install flask-login
```

在 Flask-Login 中实现的发送信号代码是：

```
user_logged_in.send(current_app._get_current_object(), user=_get_user())
```

订阅这个信号可以了。现在我们实现一个功能齐备的登录例子 (`login_signal.py`)。

初始化的部分和之前的例子很像，我们只展示不同的部分：

```
import flask_login
```

```
login_manager = flask_login.LoginManager()
login_manager.init_app(app)
```

```
password = '123' # 只要用户输入的密码是123就可以登录
```

`flask_login` 提供了 `UserMixin`，有一些用户相关的属性。

- `is_authenticated`：是否被验证。
- `is_active`：是否被激活。
- `is_anonymous`：是否是匿名用户。
- `get_id()`：获得用户的 id，并转换为 Unicode 类型。

可以在创建模型的时候继承 `UserMixin`：

```
class User(flask_login.UserMixin, db.Model):
    __tablename__ = 'login_users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    login_count = db.Column(db.Integer, default=0)
    last_login_ip = db.Column(db.String(128), default='unknown')

db.create_all()
```

添加 `user_logged_in` 信号后，一登录就会触发 `user_logged_in` 信号，增加登录次数，并添加最近登录 IP：

```
@flask_login.user_logged_in.connect_via(app)
```

```
def _track_logins(sender, user, **extra):
    user.login_count += 1
    user.last_login_ip = request.remote_addr
    db.session.add(user)
    db.session.commit()
```

使用 `user_loader` 装饰器的回调函数非常重要，它将决定 `user` 对象是否在登录状态：

```
@login_manager.user_loader
def user_loader(id):
    user = User.query.filter_by(id=id).first()
    return user
```

这个 `id` 参数的值是在 `flask_login.login_user(user)` 中传入的 `user` 的 `id` 属性。

现在添加登录的视图，如果是 GET 方法，只返回一个简单的表单：

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return ''
    <form action='login' method='POST'>
        <input type='text' name='name' id='name' placeholder='name'></input>
        <input type='password' name='pw' id='pw' placeholder='password'></input>
        <input type='submit' name='submit'></input>
    </form>
    ...

    name = request.form.get('name')
    if request.form.get('pw') == password:
        user = User.query.filter_by(name=name).first()
        if not user:
            user = User(name=name)
            db.session.add(user)
            db.session.commit()
            flask_login.login_user(user)
            return redirect(url_for('protected'))

    return 'Bad login'
```

如果表单的密码是 123，就会跳转到视图函数 `protected` 上。显示登录的信息：

```
@app.route('/protected')
@flask_login.login_required
def protected():
    user = flask_login.current_user
    return 'Logged in as: {} | Login_count: {} | IP: {}'.format(
        user.name, user.login_count, user.last_login_ip)
```


Flask 的扩展

Flask 扩展的生态非常繁荣，本节介绍其中最常用的扩展。

Flask-Script

Django 提供了如下管理命令：

```
python manage.py startapp
python manage.py runserver
```

Flask 也可以通过 Flask-Script 添加运行服务器、设置数据库、定制 shell 等功能的命令。

我们先安装它：

```
> pip install flask-script
```

借用之前的文件托管服务项目的代码：

```
> cp chapter3/section5/*.py chapter4/section2
```

现在使用 Flask-Script 管理服务（manage.py）：

```
from flask_script import Manager, Server, Shell, prompt_bool
```

```
from app import app, db, PasteFile
```

```
manager = Manager(app)
```

```
def make_shell_context():
    return {
        'db': db,
        'PasteFile': PasteFile,
        'app': app
    }
```

```
@manager.command
```

```
def dropdb():
```

```
    if prompt_bool(
```

```
        'Are you sure you want to lose all your data'):
```

```
        db.drop_all()
```

```
@manager.option('-h', '--filehash', dest='filehash')
```

```
def get_file(filehash):
    paste_file = PasteFile.query.filter_by(filehash=filehash).first()
    if not paste_file:
        print 'Not exists'
    else:
        print 'URL is {}'.format(paste_file.get_url('i'))

manager.add_command('shell', Shell(make_context=make_shell_context))
manager.add_command('runserver', Server(
    use_debugger=True, use_reloader=True,
    host='0.0.0.0', port=9000)
)

if __name__ == '__main__':
    manager.run()
```

现在就可以使用如下命令来管理了：

```
> cd chapter4/section2
> python manage.py get_file -h ec12e434b48648f0a65ac28a28759ba5.jpg # 在终端通过
    filehash获取文件路径
URL is http://localhost:9000/i/ec12e434b48648f0a65ac28a28759ba5.jpg
> python manage.py get_file -h not_exists_file
Not exists
> python manage.py dropdb # 在测试环境里可以用来清理数据
Are you sure you want to lose all your data [n]: n
> python manage.py shell # 自帶了三个内置变量的shell
In : db
Out: <SQLAlchemy engine='mysql://web:web@localhost:3306/r'>
In : PasteFile
Out: app.PasteFile
> python manage.py runserver # 通过manage.py启动服务
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
* Restarting with stat
```



在很多地方你可能都会看到使用“from flask.ext.package import X”的用法，但这种用法已经受到官方反对（<http://bit.ly/2aradPt>），要采用“from flask_package import X”的方式。

如果你已经使用 Flask 0.11，可以考虑使用 Flask 自带的命令行接口替代它。

Flask-DebugToolbar

Django 有非常知名的 Django-DebugToolbar, 而 Flask 也有对应的替代工具 Flask-DebugToolbar。它会在浏览器上添加右边栏, 可以快速查看环境变量、上下文内容, 方便调试。

我们先安装它:

```
> pip install flask-debugtoolbar
```

它的使用也很简单, 但是注意 debug 要为 True 才可以看到调试边栏 (app_debug.py):

```
from flask import Flask
from flask_debugtoolbar import DebugToolbarExtension

app = Flask(__name__)

app.debug = True
app.config['SECRET_KEY'] = 'a secret key'

toolbar = DebugToolbarExtension(app)

@app.route('/')
def hello():
    return '<body></body>' # 模板需要至少有body元素

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000, debug=app.debug)
```

Flask-DebugToolbar 内置了很多面板, 如表 4.1 所示。

表 4.1 Flask-DebugToolbar 内置的面板及功能

面 板	功 能
Versions	列出安装的包的版本
Time	显示处理当前请求花费的时间的信息
HTTP Headers	显示当前请求的 HTTP 头信息
Request Vars	显示当前请求带的变量, 包含请求参数、cookie 信息等
Config	显示 app.config 的变量值
Templates	显示模板请求参数信息
SQLAlchemy	显示当前请求下的 SQL。需要设置 SQLALCHEMY_RECORD_QUERIES 为 True
Logging	显示请求过程中的日志信息

续表

面 板	功 能
Route List	列出 Flask 的路由规则
Profiler	对当前请求添加性能分析。默认是关闭的，需要点击红色的钩，让它变成绿色

Flask-Migrate

使用关系型数据库时，修改数据库模型和更新数据库这样的工作时有发生，而且很重要。怎么做到既安全又方便呢？

SQLAlchemy 作者为此开发了迁移框架 Alembic，Flask-Migrate 就是基于 Alembic 做了轻量级封装，并集成到 Flask-Script 中。所有操作都通过 Flask-Script 命令完成。它能跟踪数据库结构的变化，把变化的部分应用到数据库中。

我们先安装它：

```
> pip install Flask-Migrate
```

定义一个 User 类（users.py）：

```
from ext import db

class User(db.Model):
    __tablename__ = 'login_users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    login_count = db.Column(db.Integer, default=0)
    last_login_ip = db.Column(db.String(128), default='unknown')
```

现在对文件托管服务添加迁移支持（app_migrate.py）：

```
from flask import Flask
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

from ext import db

app = Flask(__name__)
app.config.from_object('config')

db.init_app(app)
```

```
migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)
```

```
if __name__ == '__main__':
    manager.run()
```

我们现在想要扩充两个字段 email 和 password，流程如下。

1. 初始化迁移工作：

```
> python chapter4/section2/app_migrate.py db init
```

这会在当前目录下增加一个 migrations 目录，这个目录应该放进版本库。

2. 修改表结构，给 User 类添加 email 和 password 这两个字段：

```
email = db.Column(db.String(128), nullable=False)
password = db.Column(db.String(256), nullable=False)
```

3. 创建迁移的脚本：

```
> python chapter4/section2/app_migrate.py db migrate
INFO [alembic.runtime.migration] Context impl MySQLImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.autogenerate.compare] Detected added column 'users.email'
INFO [alembic.autogenerate.compare] Detected added column 'users.password'
INFO [alembic.autogenerate.compare] Detected NULL on column 'users.login_count'
Generating /home/ubuntu/web_develop/migrations/versions/1388a7cf600a_.py ...
done
```

这会在 migrations/versions/ 目录下添加一个执行的脚本，文件名就是版本号。版本的对应关系在当前库的 alembic_version 表中。需要注意的是，假如你的数据库里还有其他的表没有放到迁移脚本中，就会被删掉，所以 app_migrate.py 这样的管理脚本应该覆盖所有重要的表，而所有模型文件都使用“from ext import db”，就可以保证这一点。

这一步并没有实际操作数据库，所以一定要注意终端输出，确定和自己的预想一样。

4. 更新数据库：

```
> python chapter4/section2/app_migrate.py db upgrade
INFO [alembic.runtime.migration] Context impl MySQLImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Running upgrade -> f146d724a693, empty
message
```

这样就更新数据库了。

验证一下对数据库结构的修改：

```
mysql> select * from login_users;
+-----+-----+-----+-----+-----+-----+
| id | name | login_count | last_login_ip | email | password |
+-----+-----+-----+-----+-----+-----+
| 0 | a | 11 | 10.0.2.2 | | |
+-----+-----+-----+-----+-----+-----+
```

可以看到这两个字段已经出现了。是不是非常方便和安全呢？如果发现问题也可以很容易地取消更新：

```
> python chapter4/section2/app_migrate.py db downgrade
INFO [alembic.runtime.migration] Context impl MySQLImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Running downgrade 68260ed097cc -> , empty message
```

再来验证一下：

```
mysql> select * from login_users;
+-----+-----+-----+-----+
| id | name | login_count | last_login_ip |
+-----+-----+-----+-----+
| 0 | a | 11 | 10.0.2.2 |
+-----+-----+-----+-----+
```

上例中存储 IP 时使用了字符串，这只是为了展示起来更直观，更好的方法是把 IP 转换为整数。

```
In : import struct
In : import socket
In : def ip2int(addr):
...:     return struct.unpack("!I", socket.inet_aton(addr))[0]
...:
In : def int2ip(addr):
...:     return socket.inet_ntoa(struct.pack("!I", addr))
...:
In : ip2int('10.0.2.2')
Out: 167772674
```

Flask-WTF

Flask-WTF 是一个集成 WTForms 的表单验证和渲染的扩展。我们先安装它；

```
> pip install Flask-WTF
```

通过实现一个注册功能的应用来了解它（app_wtf.py）。注册表单的代码如下：

```
from flask_wtf import Form
from wtforms import TextField, PasswordField
from wtforms.validators import length, Required, EqualTo

class RegistrationForm(Form):
    name = TextField('Username', [length(min=4, max=25)])
    email = TextField('Email Address', [length(min=6, max=35)])
    password = PasswordField('New Password', [
        Required(), EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
```

应用代码如下：

```
from flask import Flask, render_template, request
from flask_wtf.csrf import CsrfProtect

from ext import db
from users import User

app = Flask(__name__, template_folder='../templates')
app.config.from_object('config')
CsrfProtect(app)
db.init_app(app)

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(name=form.name.data, email=form.email.data,
                    password=form.password.data)
        db.session.add(user)
        db.session.commit()
        return 'register succeeded!'
    return render_template('chapter4/section2/register.html', form=form)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000, debug=app.debug)
```

表单模板（register.html）使用默认的 Jinja2 语法：

```
> cat templates/chapter4/section2/register.html
{% macro render_field(field) %}
    <div class="form-group">
        <div class="form-label">{{ field.label }}</div>
        <div class="form-body">
            {{ field(**kwargs)|safe }}
        </div>

        {% if field.errors %}
        <ul class="warning">
            {% for error in field.errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
        {% endif %}
    </div>
{% endmacro %}

<form method=post action='/register'>
    {{ form.csrf_token }}
    <dl>
        {{ render_field(form.name) }}
        {{ render_field(form.email) }}
        {{ render_field(form.password) }}
        {{ render_field(form.confirm) }}
    </dl>
    <p><input type=submit value=Register>
</form>
```

这样就完成了一个带有 CSRF 保护的注册表单的功能。

Flask-Security

Flask-Security 非常强大，它提供角色管理、权限管理、用户登录、邮箱验证、密码重置、密码加密、跟踪用户登录状态等功能。先安装它：

```
> pip install flask-security
```

Flask-Security 提供了 7 种基本模板。如果想要定制模板，可以在应用的模板目录下创建名为 security 的目录，添加对应名字的模板。这 7 种模板类型和用途如表 4.2 所示。

表 4.2 7 种基本模板和用途

模 板	功 能
security/forgot_password.html	忘记密码
security/login_user.html	用户登录
security/register_user.html	用户注册
security/reset_password.html	重置密码
security/change_password.html	改变密码
security/send_confirmation.html	发送确认信息
security/send_login.html	无密码方式

可以指定对应的变量替换模板，下面的例子中将使用其他的登录模板，也就是要指定变量 SECURITY_LOGIN_USER_TEMPLATE。

Flask-Security 还提供了 8 种上下文处理的装饰器，类似于钩子。这 8 种处理器类型如表 4.3 所示。

表 4.3 8 种处理器及其功能

处 理 器	功 能
context_processor	所有视图都会触发
forgot_password_context_processor	忘记密码时视图就会触发
login_context_processor	登录时视图会触发
register_context_processor	注册时视图会触发
reset_password_context_processor	重置密码时视图会触发
change_password_context_processor	改变密码时视图会触发
send_confirmation_context_processor	发送确认信息时视图会触发
send_login_context_processor	无密码方式登录时视图会触发

Flask-Security 也提供了 8 种表单，表单作用如表 4.4 所示。

表 4.4 8 种表单及其用途

表 单	用 途
login_form	登录
confirm_register_form	注册确认

续表

表 单	用 途
register_form	注册
forgot_password_form	忘记密码
reset_password_form	重置密码
change_password_form	改变密码
send_confirmation_form	发送确认信息
passwordless_login_form	无密码登录

在看例子之前，先了解一个新的概念：“角色”。角色定义了用户的类型。如果一个站点功能很少，只需要普通用户和管理员两种权限类型就可以了。但随着业务的扩展，用户具有的特殊的权限类型越来越多，对于权限管理而言，不能为每个人都授予管理员这样的角色，这个时候就需要实现多种类型的角色权限，不同的角色甚至可以具备多种角色权限。

我们使用位运算做权限控制。位运算在 Linux 文件系统上就有体现，一个用户对文件或目录所拥有的权限分为三种：“可读（1）”、“可写（2）”和“可执行（4）”，它们之间可以任意组合：有可读和可写权限就用 3 来表示（ $1+2=3$ ）；有可读和可执行权限就用 5 来表示（ $1+4=5$ ），三种权限全部拥有就用 7 表示（ $1+2+4=7$ ）。为什么选择 1、2、4 这样的有规律的数据呢？先看看下面的例子：

```
In : int('00000001', 2)
Out: 1
In : int('00000010', 2)
Out: 2
In : int('00000100', 2)
Out: 4
```

通过标志位判断是否有权限，如果有权限，对应位就置 1，比如三种权限都有，就是：

```
In : int('00000111', 2)
Out: 7
```

它其实就是对三个二进制位做按位或计算得到的：

```
In : int('00000100', 2) | int('00000010', 2) | int('00000001', 2)
Out: 7
```

判断权限的时候就把这个值（假设叫作 A）和要判断的权限（叫作 B）做按位与计算，这样就把它们中都为 1 的标志位置为 1，如果结果还等于 B，就说明它有这样的权限，否则说明对应的标志位没有置 1，没有权限：

```
In : int('00000111', 2) & int('00000001', 2) == int('00000001', 2)
Out: True
```

```
In : int('00000100', 2) & int('00000001', 2) == int('00000001', 2)
Out: False
```

在这里，权限定义如下：

```
class Permission(object):
    LOGIN = 0x01
    EDITOR = 0x02
    OPERATOR = 0x04
    ADMINISTER = 0xff
    PERMISSION_MAP = {
        LOGIN: ('login', 'Login user'),
        EDITOR: ('editor', 'Editor'),
        OPERATOR: ('op', 'Operator'),
        ADMINISTER: ('admin', 'Super administrator')
    }
```

其中 ADMINISTER 拥有全部权限，使用 0xff 这个最大值。

Flask-Security 支持 SQLAlchemy、MongoEngine 和 Peewee 定义模型。现在基于 Flask-SQLAlchemy 和之前的文件托管服务来实现一个简单的应用（app_security.py）。

先定义角色模型：

```
from flask_security import RoleMixin
```

```
class Role(db.Model, RoleMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255), unique=True)
    permissions = db.Column(db.Integer, default=Permission.LOGIN)
    description = db.Column(db.String(255))
```

给 User 这个模型添加 roles 字段，指向模型 Role，并且添加判断权限的方法：

```
from functools import reduce
from operator import or_
```

```
from flask_security import UserMixin
```

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    password = db.Column(db.String(255))
```

```
active = db.Column(db.Boolean())
confirmed_at = db.Column(db.DateTime())
roles = db.relationship('Role', secondary=roles_users,
                        backref=db.backref('users', lazy='dynamic'))

def can(self, permissions):
    if self.roles is None:
        return False
    all_perms = reduce(or_, map(lambda x: x.permissions, self.roles))
    return all_perms & permissions == permissions

def can_admin(self):
    return self.can(Permission.ADMINISTER)
```

用户和角色是多对多的关系，需要定义一个用于关系的辅助表。对于这个辅助表，强烈建议不使用模型，而是采用一个实际的表：

```
roles_users = db.Table(
    'roles_users',
    db.Column('user_id', db.Integer,
              db.ForeignKey('user.id')),
    db.Column('role_id', db.Integer, db.ForeignKey('role.id')))
```

上面的 `db.relationship` 还使用了 `backref`，它表示反向引用。举个例子：

```
In : r = Role()
In : r.name = 'role_1'
In : r.permissions = Permission.LOGIN
In : r.description = 'test role'

In : u = User()
In : u.email = '1@qq.com'
In : u.password = '123'
In : u.confirmed_at = datetime.now()
In : u.roles = [r]

In : db.session.add(r)
In : db.session.add(u)
In : r.users
Out: <sqlalchemy.orm.dynamic.AppenderBaseQuery at 0x7f6548084dd0>
In : db.create_all()
In : r.users.all()
Out: [<__main__.User at 0x7f65480c2e90>]
```

Role 对象通过 `users` 就能反向获取有对应权限的用户列表了。

接着添加 login_context_processor 钩子：

```
user_datastore = SQLAlchemyUserDatastore(db, User, Role)
security = Security(app, user_datastore, register_form=LoginForm)
```

```
@security.login_context_processor
def security_login_processor():
    print 'Login'
    return {}
```

这个例子中，我们通过添加 before_first_request 钩子实现初始化：

```
@app.before_first_request
def create_user():
    db.drop_all()
    db.create_all()

    for permissions, (name, desc) in Permission.PERMISSION_MAP.items():
        user_datastore.find_or_create_role(
            name=name, description=desc, permissions=permissions)
    for email, passwd, permissions in (
        ('dongwm@dongwm.com', '123', (
            Permission.LOGIN, Permission.EDITOR)),
        ('admin@dongwm.com', 'admin', (Permission.ADMINISTER,))):
        user_datastore.create_user(email=email, password=passwd)
        for permission in permissions:
            user_datastore.add_role_to_user(
                email, Permission.PERMISSION_MAP[permission][0])
    db.session.commit()
```

每次在第一次接收请求的时候就会删除相关表，再重新创建这些表，并创建两个用户，用户权限分别如下。

- dongwm@dongwm.com：它具有 LOGIN 与 EDITOR 两种权限，但有些页面访问不了。
- admin@dongwm.com：管理员，拥有全部的权限。

然后添加验证访问权限的装饰器：

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def _deco(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)
        return _deco
```

```
return decorator
```

```
def admin_required(f):
    return permission_required(Permission.ADMINISTER)(f)
```

`current_user` 就是一个 `User` 对象，通过 `User` 类添加的 `can` 方法判断权限。

最后看一下给视图添加权限控制的方法：

```
@app.route('/')
@login_required
@permission_required(Permission.LOGIN)
def index():
    return 'Login in'

@app.route('/admin/')
@login_required
@admin_required
def admin():
    return 'Only administrators can see this!'
```

上例使用了自定义的登录模板和自定义的处理器，当然还可以自定义表单。通过 `dongwm@dongwm.com` 登录后是看不到 `/admin/` 页面的，而用 `admin@dongwm.com` 就可以看到全部页面。

自定义的登录模板（`login_user.html`）内容如下：

```
{% from "security/_macros.html" import render_field_with_errors, render_field %}
{% include "security/_messages.html" %}
<h1>Custom Login</h1>
<form action="{{ url_for_security('login') }}" method="POST" name="login_user_form">
    {{ login_user_form.hidden_tag() }}
    {{ render_field_with_errors(login_user_form.email) }}
    {{ render_field_with_errors(login_user_form.password) }}
    {{ render_field_with_errors(login_user_form.remember) }}
    {{ render_field(login_user_form.next) }}
    {{ render_field(login_user_form.submit) }}
</form>
{% include "security/_menu.html" %}
```



`User` 类中的 `password` 字段使用了明文存储，这是为了让例子更清晰，生产环境中请勿使用明文存储重要的内容。下一节我们将介绍加密方法。

Flask-RESTful

Flask-RESTful 帮助你快速创建 REST API 服务。先安装它：

```
> pip install flask-restful
```

现在实现一个能创建、查询和删除用户的 API 例子（app_restful.py）：

```
from flask import Flask, request
from flask_restful import Resource, Api, reqparse, fields, marshal_with
from flask_sqlalchemy import SQLAlchemy
```

```
app = Flask(__name__)
app.config.from_object('config')
api = Api(app)
db = SQLAlchemy(app)
```

```
parser = reqparse.RequestParser()
parser.add_argument('admin', type=bool, help='Use super manager mode',
                    default=False)
```

```
resource_fields = {
    'id': fields.Integer,
    'name': fields.String,
    'address': fields.String
}
```

```
class User(db.Model):
    __tablename__ = 'restful_user'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    address = db.Column(db.String(128), nullable=True)
```

```
db.create_all()
```

```
class UserResource(Resource):
    @marshal_with(resource_fields)
    def get(self, name):
        user = User.query.filter_by(name=name).first()
        return user

    def put(self, name):
        address = request.form.get('address', '')
```

```

        user = User(name=name, address=address)
        db.session.add(user)
        db.session.commit()
        return {'ok': 0}, 201

    def delete(self, name):
        args = parser.parse_args()
        is_admin = args['admin']
        if not is_admin:
            return {'error': 'You do not have permissions'}
        user = User.query.filter_by(name=name).first()
        db.session.delete(user)
        db.session.commit()
        return {'ok': 0}

```

```
api.add_resource(UserResource, '/users/<name>')
```

```

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000, debug=True)

```

在 Flask-RESTful 中，一个地址下的数据称为资源（Resource）。装饰器 `marshal_with` 做了把模型实例的属性组合成一个字典的抽象工作。

在终端上验证效果：

```

> http -f put http://localhost:9000/users/xiaoming address='Beijing'
HTTP/1.0 201 CREATED
Content-Length: 16
Content-Type: application/json
Date: Sun, 29 May 2016 04:01:49 GMT
Server: Werkzeug/0.11.10 Python/2.7.11+

```

```

{
    "ok": 0
}

```

```

> http -f put http://localhost:9000/users/wanglang --print b
{
    "ok": 0
}

```

```

> http -f get http://localhost:9000/users/xiaoming --print b
{
    "address": "Beijing",
    "id": 1,

```



```
"name": "xiaoming"
}

> http -f delete http://localhost:9000/users/xiaoming --print b
{
  "error": "You do not have permissions"
}

> http -f delete http://localhost:9000/users/xiaoming admin=1 --print b
{
  "ok": 0
}

> http -f get http://localhost:9000/users/xiaoming --print b
{
  "address": null,
  "id": 0,
  "name": null
}
```

Flask-Admin

有了 Flask-Admin 的帮助，我们用很少的代码就能像 Django 那样实现一个管理后台。它支持 Pymongo、Peewee、Mongoengine、SQLAlchemy 等数据库使用方法，自带了基于模型的数据管理、文件管理、Redis 的页面命令行等类型后台。尤其是模型的管理后台，甚至可以细粒度定制字段级别的权限。

我们先安装它：

```
> pip install Flask-Admin
```

现在基于 Flask-Login 和 Flask-SQLAlchemy 实现包含如下功能的后台（app_admin.py）：

- 可以在后台操作数据库中的数据。
- 静态文件管理。
- 在导航栏添加一些链接和视图，比如笔者的 GitHub 地址、Google 链接以及回首页的链接。还添加一个动态的链接，点击它可以登录和退出。当登录后会动态地添加一个“Authenticated”的链接。
- 自定义点击“Authenticated”的链接后看到的模板。

第一步还是定义 User 模型，本例借用 users.py 里面的 User，再继承 UserMixin 即可：

```
from ext import db
from users import User as _User
```

```
class User(_User, UserMixin):
    pass
```

登录管理器和 4.2.6 节介绍的 Flask-Login 信号的用法相同，这里就不展示了。

接着添加主页、登录和退出的视图：

```
from flask_login import login_user, logout_user
```

```
USERNAME = 'xiaoming'
```

```
@app.route('/')
def index():
    return '<a href="/admin/">Click me to get to Admin!</a>'
```

```
@app.route('/login/')
def login_view():
    user = User.query.filter_by(name=USERNAME).first()
    login_user(user)
    return redirect(url_for('admin.index'))
```

```
@app.route('/logout/')
def logout_view():
    logout_user()
    return redirect(url_for('admin.index'))
```

这样的视图只作为管理后台的可点击链接来用：

```
from flask_admin import Admin
from flask_admin.base import MenuLink
```

```
class AuthenticatedMenuLink(MenuLink):
    def is_accessible(self):
        return current_user.is_authenticated
```

```
class NotAuthenticatedMenuLink(MenuLink):
```

```
def is_accessible(self):
    return not current_user.is_authenticated
```

```
admin = Admin(app, name='web_develop', template_mode='bootstrap3')
admin.add_link(NotAuthenticatedMenuLink(name='Login',
                                         endpoint='login_view'))
admin.add_link(AuthenticatedMenuLink(name='Logout',
                                       endpoint='logout_view'))
```

也可以直接使用 url 参数指定地址：

```
admin.add_link(MenuLink(name='Back Home', url='/'))
admin.add_link(MenuLink(name='Google', category='Links',
                        url='http://www.google.com/'))
admin.add_link(MenuLink(name='GitHub', category='Links',
                        url='https://github.com/dongweiming'))
```

其中 category 会创建一个叫作 Links 的下拉菜单，把 Google 和 GitHub 链接放进去。

在 Flask-Admin 中指定视图需要继承它提供的 BaseView，或者使用 contrib 中自带的视图类，比如 FileAdmin 和 ModelView：

```
from flask_admin.base import BaseView, expose
from flask_admin.contrib.sqla import ModelView
from flask_admin.contrib.fileadmin import FileAdmin

class MyAdminView(BaseView):
    @expose('/')
    def index(self):
        return self.render('chapter5/section2/authenticated-admin.html')

    def is_accessible(self):
        return current_user.is_authenticated

admin.add_view(ModelView(User, db.session))

path = os.path.join(os.path.dirname(__file__), '../static')
admin.add_view(FileAdmin(path, '/static/', name='Static Files'))

# 创建一个名为Authenticated的链接，但是必须登录才能访问
admin.add_view(MyAdminView(name='Authenticated'))
```

其中有如下细节需注意。

- MyAdminView 定义的首页地址没有验证是不能访问的。
- 上面提到的视图可以通过设置 endpoint 参数自定义链接，比如 “ModelView(User, db.session)” 生成的子路径是 “/admin/user”，修改为：“ModelView(User, db.session, endpoint='new_user')”，就可以使用 “/admin/new_user” 来访问了。
- 提到的 authenticated-admin.html 模板就是我们自定义的点击 Authenticated 的链接后看到的模板。它的内容如下：

```
{% extends 'admin/master.html' %}
{% block body %}
    Hello World from Authenticated Admin!
{% endblock %}
```

最后使用 before_first_request 钩子初始化数据库：

```
@app.before_first_request
def create_user():
    db.drop_all()
    db.create_all()

    user = User(name=USERNAME, email='a@dongwm.com', password='123')
    db.session.add(user)
    db.session.commit()
```

本例可以使用 xiaoming 这个固定的用户来执行登录、退出等操作。

添加管理后台后，当访问 http://localhost:9000/ 的时候，将显示 index 函数的返回内容。访问 http://localhost:9000/admin/ 就可以看到未登录的管理后台了，页面只有一个菜单栏，如图 4.1 所示。



图 4.1 未登录的管理后台

其中：

- User 这个链接是通过 ModelView 实现的，也就是在后台操作 User 表。操作功能包含修改、创建、删除等。
- Static Files 这个链接是通过 FileAdmin 实现的，它可以管理项目的静态文件。操作功能包含重命名、上传、查看和删除文件，创建目录等。
- Links 下拉菜单中包含了如作者的 GitHub 地址、Google 链接等选项。

- 最后一个是 Login，这个按钮链接是动态的，登录前显示“Login”，登录后显示“Logout”。当登录后会菜单栏会动态添加一个“Authenticated”的链接。
- 登录后点击 Authenticated 的链接可以看到 authenticated-admin.html 模板渲染的内容，如果未登录就访问这个链接地址，则返回 403 错误。

Flask-Assets

Web 网站的 Javascript 和 CSS 源文件很多。举个例子，某应用的结构如下：

```
> cd ~/web_develop/chapter4/section2
> tree static
static
├── css
│   ├── all.min.css
│   ├── base-min.css
│   └── buttons-min.css
├── js
│   ├── all.min.js
│   ├── src
│   │   ├── click.es6
│   │   ├── common.js
│   │   └── highlight.js
│   └── vendor
│       └── jquery.min.js
└── scss
    ├── common.scss
    └── forms.scss
```

static 目录下有 3 个文件夹。

- css：存放了外部的 CSS 库和一些已压缩的 CSS 文件。
- scss：存放组件 SCSS 源文件。
- js：存放外部的 JavaScript 库和其他 JavaScript 文件。其中 src 子目录下存放使用原生 JavaScript 和 ES6 这两种方式编写的源代码，通过后缀名来分辨类型。

ES6 是 ECMAScript6 的简称，也称为 ES2015，它是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标是让 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。现在很多互联网公司都已经开始使用 ES6、React 等技术重构已有项目或者在新的项目中使用，所以本节也演示一下如何在 Flask-Assets 中使用 ES6。由于现在用户浏览器还没有完全支持 ES6，所以需要借助 Babel 这个转码器，将 ES6 代码转为 ES5 代码，从而支持大多数 JavaScript 环境。

首先安装 Babel 和对应的预设：

```
> sudo apt-get install npm -yq
> sudo npm install --global babel-cli
> sudo ln -s `which nodejs` /usr/bin/node
> npm install babel-preset-es2015
```

其中的 babel-preset-es2015 就是预设。默认的 Babel 还没有“翻译”代码的能力，而仅仅把代码从一处拷贝到另一处。需要通过这样的预设（presets，也就是一组插件）来指示 Babel 去“翻译”ES6 代码。

如上例，前端资源会非常多，尤其是组件化开发的方式下。不能在模板中把这些组件挨个引用进来，因为这会增加网络延时，造成页面访问很慢。常见的解决方法是合并和压缩。Flask-Assets 是 webassets 库的 Flask 扩展，它提供了更简单的使用 webassets 的方式。由于 webassets 在添加 Babel 支持之后并没有发布新版本，需要克隆项目安装 webassets 之后再安装 Flask-Assets：

```
> git clone https://github.com/miracle2k/webassets
> cd webassets
> python setup.py install
> pip install flask_assets
```

webassets 自带的过滤器很多，主要包含 4 种类型，如表 4.5 所示。

表 4.5 webassets 自带的过滤器

过 滤 器	用 途
Babel	用来翻译 ES6 代码
JavaScript 压缩	常用的有 rjsmin、yui_js（也就是雅虎的 YUI Compressor）、jsmin 和 closure_js
CSS 压缩	常用的有 cssmin、yui_css 和 cleancss
编译器	常用的有 less、sass、pyscss、libsass、stylus 和 typescript

我们选择 jsmin 作为 JavaScript 压缩工具，cssmin 作为 CSS 压缩工具，pyscss 作为 CSS 编译工具。先安装它们：

```
> pip install jsmin cssmin pyscss
```

为了方便管理，把合并、压缩的内容单独存放（assets.py）：

```
from webassets.filter import get_filter
from flask_assets import Environment, Bundle
```

```
css_common = Bundle('scss/common.scss',
                    filters='pyscss', output='css/common.css',
                    debug=False)

css_all = Bundle('css/base-min.css', css_common,
                'css/buttons-min.css',
                filters='cssmin', output='css/all.min.css')

js_common = Bundle('js/src/*.js', output='js/common.js')

es2015 = get_filter('babel', presets='es2015')
es2015_all = Bundle('js/src/*.es6', output='js/es6.js', filters=es2015)

js_all = Bundle(
    'js/vendor/jquery.min.js',
    js_common, es2015_all,
    filters='jsmin', output='js/all.min.js')

def init_app(app):
    webassets = Environment(app)
    webassets.register('css_all', css_all)
    webassets.register('js_all', js_all)
    webassets.manifest = 'cache' if not app.debug else False
    webassets.cache = not app.debug
    webassets.debug = app.debug
```

其中，Bundle 既可以接受纯字符串的路径，也接受其他的 Bundle 实例。

对模块 assets 添加 init_app 函数是为了使其更像一个 Flask 扩展（app_assets.py）：

```
from flask import Flask, render_template

import assets

app = Flask(__name__)
assets.init_app(app)

@app.route('/')
def hello():
    return render_template('index.html')
```

在 init_app 函数中注册了 css_all 和 js_all 这两个资源，需要在模板中使用如下 Jinja2 语法来引用：

```
{% assets "css_all" -%}  
    <link rel="stylesheet" href="{{ ASSET_URL }}">  
{%- endassets %}
```

Werkzeug 的使用

Werkzeug 是 WSGI 协议层工具集，本节我们来挖掘 Werkzeug 中有用的函数和类，帮助我们进行 Web 开发。它可以独立安装：

```
> pip install Werkzeug
```

DebuggedApplication

werkzeug.debug.DebuggedApplication 实现了杀手级的 Debug 调试器，可以用如下方式启用调试器：

```
from werkzeug.debug import DebuggedApplication  
from myapp import app  
app = DebuggedApplication(app, evalex=True)
```

如果在 Django 中使用它，可以通过 django-extensions 的 runserver_plus (<http://bit.ly/28Rpopa>) 看一下 Tornado 框架集成 DebuggedApplication 的例子 (app_tornado.py)。

首先安装 Tornado：

```
> pip install tornado
```

然后自定义请求处理类：

```
import tornado.web  
from werkzeug.debug import DebuggedApplication  
  
class Handler(tornado.web.RequestHandler):  
    def initialize(self, debug):  
        if debug:  
            self.write_error = self.write_debugger_error  
  
    def write_debugger_error(self, status_code, **kwargs):  
        assert isinstance(self.application, DebugApplication)  
        html = self.application.render_exception()  
        self.write(html.encode('utf-8', 'replace'))
```


如果是开启 DEBUG 模式，则使用 `DebugApplication` 的 `render_exception` 方法生成 HTML。基于 `Handler` 创建一个使用 GET 就会报错的 `BadHandler`：

```
class BadHandler(Handler):
    def get(self):
        raise Exception('This is a test')
        self.write('You will never see this text.')
```

`DebugApplication` 类的内容如下：

```
import tornado.wsgi
from tornado.web import Application
from werkzeug.debug.tbtools import get_current_traceback

class RequestDispatcher(tornado.web._RequestDispatcher):
    def set_request(self, request):
        super(RequestDispatcher, self).set_request(request)
        if '__debugger__' in request.uri:
            return self.application.debug_container(request)

class DebugApplication(Application):
    def __init__(self, *args, **kwargs):
        super(DebugApplication, self).__init__(*args, **kwargs)
        self.debug_app = DebuggedApplication(self, evalex=True)
        self.debug_container = tornado.wsgi.WSGIContainer(self.debug_app)

    def start_request(self, server_conn, request_conn):
        return RequestDispatcher(self, request_conn)

    def render_exception(self):
        traceback = get_current_traceback()

        for frame in traceback.frames:
            self.debug_app.frames[frame.id] = frame
            self.debug_app.tracebacks[traceback.id] = traceback

        return traceback.render_full(evalex=True,
                                     secret=self.debug_app.secret)
```

创建的路由统一放在函数中，虽然仅有 `/error/` 这一个可用地址，但是更利于独立管理：

```
def create_application(debug=False):
    handlers = [
        ('/error/', BadHandler, {'debug': debug}),
    ]
    if debug:
```

```
        return DebugApplication(handlers, debug=True)
    return Application(handlers, debug=debug)
```

主函数如下：

```
import tornado.ioloop
from tornado.options import define, options, parse_command_line

define('debug', default=False, type=bool, help='Run in debug mode.')
define('port', default=9000, type=int, help='Port on which to listen.')
parse_command_line()

logger = logging.getLogger()
port = options.port
application = create_application(debug=options.debug)
logger.info('Running tornado on port {}'.format(port))
application.listen(port)
tornado.ioloop.IOLoop.instance().start()
```

使用 `define` 可以定义命令行参数的名字、类型和默认值。`create_application` 中的逻辑是，当 `debug` 为 `False`，也就是不指定 `debug` 参数时：

```
> python chapter4/section3/app_tornado.py
```

则使用默认的 `Application`。

如果开启 `debug` 模式：

```
> python chapter4/section3/app_tornado.py --debug
```

则使用 `DebugApplication`，即可以使用更友好的 `werkzeug.debug.DebuggedApplication` 了。

数据结构

Werkzeug 中提供了多种定制的数据结构，在工作中有时也会需要这样的数据结构，本节列出常用的到数据结构。

1. `TypeConversionDict`：它继承于 `dict`，执行 `get` 方法的可以指定值的类型。

```
In : from werkzeug.datastructures import TypeConversionDict
In : d = TypeConversionDict(foo='42', bar='blub')
In : d.get('foo', type=int)
Out: 42
In : d.get('bar', -1, type=int)
Out: -1
```

2. ImmutableTypeConversionDict: 不可变的 TypeConversionDict。
3. MultiDict: 它继承于 TypeConversionDict, 可以对相同的键传入多个值, 会把这些值都保留下来。

```
In : from werkzeug.datastructures import MultiDict
In : d = MultiDict([('a', 'b'), ('a', 'c')])
In : d.getlist('a')
Out: ['b', 'c']
In : list(d.iterlists())
Out: [('a', ['b', 'c'])]
In : d.setlist('d', ['e', 'f'])
In : d
Out: MultiDict([('a', 'b'), ('a', 'c'), ('d', 'e'), ('d', 'f')])
In : d.poplist('d')
Out: ['e', 'f']
In : d.get('a')
Out: 'b'
```

4. ImmutableMultiDict: 不可变的 MultiDict。
5. OrderedMultiDict: 它继承于 MultiDict, 但是保留了字典的顺序。
6. ImmutableOrderedMultiDict: 不可变的 OrderedMultiDict。

功能函数

Werkzeug 中提供了多个有用的函数。

1. cached_property: 非常知名的装饰器, 它通过描述符把方法执行的结果作为一个属性 (property) 缓存下来。

```
In : class Foo(object):
....:     @cached_property
....:     def bar(self):
....:         print 'calculate something'
....:         return 1
....:

In : foo = Foo()

In : foo.bar
calculate something
Out: 1
In : foo.bar # 方法只执行了一次, 此后都直接返回结果
Out: 1
```

2. `import_string`: 一个帮助我们直接通过字符串找到对应模块的功能函数。

```
In : import_string('os')
Out: <module 'os' from '/home/ubuntu/web_develop/venv/lib/python2.7/os.pyc'>
In : import_string('werkzeug.utils')
Out: <module 'werkzeug.utils' from '/home/ubuntu/web_develop/venv/local/lib/
python2.7/site-packages/werkzeug/utils.pyc'>
```

3. `secure_filename`: 返回一个安全版本的文件名。

```
In : secure_filename('My cool movie.mov')
Out: 'My_cool_movie.mov'
In : secure_filename(' ../../etc/passwd')
Out: 'etc_passwd'
In : secure_filename(u'i contain cool \xfcml\xe4uts.txt')
Out: 'i_contain_cool_umlauts.txt'
```

密码加密

数据库中的重要字段（如密码）不能明文存储，需要加密之后存储。Web 开发常用到的方法是加盐哈希加密，也就是在加密时混入一段随机字符串（盐值，salt）再进行哈希加密（如 MD5、SHA1 等），这样即使密码相同，如果混入的盐值不同，那么哈希值也是不一样的。Werkzeug 中提供了密码加盐的哈希函数：

```
In : from werkzeug.security import generate_password_hash
In : pw_1 = generate_password_hash('xiaoming')
In : pw_2 = generate_password_hash('xiaoming')
In : pw_1
Out: 'pbkdf2:sha1:1000$ynvwQwYK$1854dfb3b61124972d68bc8d7291d333bcf583de'
In : pw_2
Out: 'pbkdf2:sha1:1000$SVT1AYa8$fdb43e5afc70f6751b11c5479e56632eab0cdf8f'
```

哈希之后的哈希字符串格式是“method\$salt\$hash”，其中 1000 表示迭代次数，默认是 1000。由于盐值是随机的，同一个密码生成的哈希值不一样，因而不容易被暴力破解。

经过哈希之后的字符串是不能获取原密码的，只能使用 `check_password_hash` 来验证：

```
In : check_password_hash(pw_1, 'xiaoming')
Out: True
In : check_password_hash(pw_2, 'xiaoming')
Out: True
In : check_password_hash(pw_1[:-1] + 'a', 'xiaoming')
Out: False
```

我们看一下 SQLAlchemy 记录密码的哈希值的方法：

```
from sqlalchemy.ext.hybrid import hybrid_property

class User(db.Model):
    __tablename__ = 'hashed_users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    _password = db.Column(db.String(256), nullable=False)

    def __init__(self, name, password):
        self.name = name
        self.password = password

    @hybrid_property
    def password(self):
        return self._password

    @password.setter
    def _set_password(self, plaintext):
        self._password = generate_password_hash(plaintext)

    def verify_password(self, password):
        return check_password_hash(self.password, password)
```

其中装饰器 `hybrid_property` 把 `password` 变成了一个混合属性，可以通过 `user.password` 属性来访问哈希的密码，也会在给 `user.password` 赋值的时候触发 `password.setter`。

看一下效果：

```
In : user = User(name='xiaoming', password='123')
In : db.session.add(user)
In : db.session.commit()
In : user.password
Out: u'pbkdf2:sha1:1000$ekbscfaz$7473514367759bc2746532db99d82598a10bf97b'
In : user.verify_password('223')
Out: False
In : user.verify_password('123')
Out: True
```

中间件

中间件（Middleware）会对每次请求添加额外的处理。可以用来记录日志、会话管理、请求验证、性能分析等工作。Werkzeug 中提供了 10 个中间件，之前提到的 `werkzeug.debug.DebuggedApplication` 也是一个中间件。下面介绍 5 个常用的中间件。

SharedDataMiddleware

一般而言，静态文件都应该使用 Nginx 来服务，但是在测试环境中或者对资源响应要求不高时，也可以使用 SharedDataMiddleware 来提供这样的服务，之前实现的文件托管服务也使用了它。我们看一个例子（app_share_middleware.py）：

```
import os

from flask import Flask
from werkzeug.wsgi import SharedDataMiddleware

app = Flask(__name__)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/static/': os.path.join(os.path.dirname(__file__), 'static')
})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

假设程序所在目录的 static 子目录下有一个叫作 main.js 的文件，可以使用 <http://localhost:9000/static/main.js> 访问。

ProfilerMiddleware

可以很方便地使用 ProfilerMiddleware 添加性能分析。当请求页面的时候，就可以获得分析的结果（app_profiler_middleware.py）：

```
from flask import Flask
from werkzeug.contrib.profiler import ProfilerMiddleware

app = Flask(__name__)
app.wsgi_app = ProfilerMiddleware(app.wsgi_app, profile_dir='/tmp')

@app.route('/')
def hello():
    return 'Hello'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

访问页面之后，结果被保存下来：

```
> ls /tmp/*.prof
/tmp/GET.root.000000ms.1464519481.prof
```

如果不指定 `profile_dir`，会在终端输出分析结果。我们来看一部分终端输出：

```
> python chapter4/section3/app_profiler_middleware.py
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
```

```
-----
PATH: '/'
      319 function calls in 0.001 seconds

Ordered by: internal time, call count

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      6   0.000    0.000    0.000    0.000 /home/ubuntu/.virtualenvs/local/lib/
python2.7/site-packages/werkzeug/local.py:160(top)
      2   0.000    0.000    0.000    0.000 /home/ubuntu/.virtualenvs/local/lib/
python2.7/site-packages/werkzeug/datastructures.py:860(
_unificodify_header_value)
     10   0.000    0.000    0.000    0.000 /home/ubuntu/.virtualenvs/local/lib/
python2.7/site-packages/werkzeug/local.py:68(__getattr__)
```

DispatcherMiddleware

`DispatcherMiddleware` 是可以调度多个应用的中间件。在第3章3.1节的时候我们实现了一个 `JSONResponse`，现在利用 `JSONResponse` 实现如下功能：

- 当访问以 `/json` 开头的地址时都默认自动用 `jsonify` 格式化。
- 访问其他地址不受影响。

代码如下（`app_dispatcher_middleware.py`）：

```
from collections import OrderedDict

from flask import Flask, jsonify
from werkzeug.wrappers import Response
from werkzeug.wsgi import DispatcherMiddleware

app = Flask(__name__)
json_page = Flask(__name__)

class JSONResponse(Response):
    @classmethod
```

```
def force_type(cls, rv, environ=None):
    if isinstance(rv, dict):
        rv = jsonify(rv)
    return super(JSONResponse, cls).force_type(rv, environ)

json_page.response_class = JSONResponse

@json_page.route('/hello/')
def hello():
    return {'message': 'Hello World!'}

@app.route('/')
def index():
    return 'The index page'

app.wsgi_app = DispatcherMiddleware(app.wsgi_app, OrderedDict((
    ('/json', json_page),
)))

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

使用这个中间件对访问地址是有副作用的,“@json_page.route('/')”等价于“@app.route('/json/')”,也就是子路径的地址前面是有/json 前缀的,这一点比较隐晦。

第5章

REST 和 Ajax

互联网公司面临多重挑战：一方面智能手机、平板电脑等移动设备层出不穷，需要支持这些设备；另一方面无论是公司内部的数据互通，还是和外部合作，都需要展示相同的商业逻辑。越来越多的公司让服务端所有商业逻辑都以 RESTful API 的方式暴露给客户端，浏览器用户可以使用 Ajax、HTML 5 等技术，通过 HTTP 的方式与后台直接交互。这种统一的机制，既减少了开发复杂度，又具有非常好的扩展性。

本章会介绍 REST 和 Ajax 相关的内容：

- 理解 REST。
- 学习如何设计一个合理、好用、符合标准的 API。
- 使用 jQuery 和 fetch 分别完成一个前后端交互的 Ajax 应用。

什么是 REST

REST 是 Roy Thomas Fielding 在他 2000 年的博士论文（<http://bit.ly/28RW8yv>）中提出的，Fielding 将他对互联网软件的架构原则，定名为 REST，即 Representational State Transfer（常见的翻译是“表现层状态转化”）的缩写。

REST 其实省略了主语：资源，表现层实际上指的是资源的表现层。资源是指 Web 上一切可识别、可命名、可找到并被处理的实体。比如 HTML 页面、音频文件、图片等。用一个 URI（统一资源定位符）指向资源，使用 HTTP 请求方法操作资源。URI 可进一步划分为统一资源名 (URN，代表资源的名字) 和统一资源定位符 (URL，代表资源的地址)，其中 URL 可以定位 HTTP 网址、FTP 服务器和文件路径等，符合绝大多数场景，所以一般都可以用 URL

替代 URI。

REST 架构风格最重要的架构约束有如下 5 个。

1. 客户端-服务器端。这种 Client/Server 的架构形式提供了基本的分布式, 客户端发起请求, 服务端决定响应或者拒绝请求, 如果出错则返回错误信息, 由客户端处理异常。
2. 无状态。通信的会话状态应该全部由客户端负责维护, 也就是请求中包含了全部必要的信息。如果使用基于服务器端的会话, 要么需要保证指定会话会使用同一个服务器响应所有请求, 要么得创建一个可供所有服务器访问的公用的会话存储区, 对每个请求都额外访问这个集中式的数据存储区获得会话状态。
3. 缓存。无状态就表示可能出现重复的请求, 事实上这些请求只需要第一次真正的执行, 其余的请求都可以享用这个已完成的结果而直接响应, 所以缓存可以抵消一部分无状态带来的影响。
4. 统一接口。统一接口意味着每个 REST 应用都共享一种通用架构, 那些熟悉这种架构的人一眼就能看明白接口的意义, 并会继续延承下去。
5. 分层系统。将系统划分为几个部分, 每个部分负责一部分相对单一的职责, 然后通过上层对下层的依赖和调用组成一个完整的系统。通常可以划分为如下三层。
 - 应用层: 负责返回 JSON 数据和其他业务逻辑。
 - 服务层: 为应用层提供服务支持, 如全站的账号系统、以及本书实现的文件托管服务等。
 - 数据访问层: 提供数据访问和存储的服务, 如数据库、缓存系统、文件系统、搜索引擎等。

REST 就是这一系列设计约束的集合, 如果一个架构符合 REST 原则, 就称它为 RESTful 架构。

RESTful API 设计指南

API 一旦发布其结构将很难修改, 因此设计和实现一个符合规范、灵活、友好的 API, 是一件非常重要的事情。本节我们来聊聊 API 设计上一些被忽略的细节、使用时的误区和一些实践。

使用名词来表示资源

URI 不应该包含动词。动词应该通过不同的 HTTP 方法来体现, 如下是几种常见的错误用法:

```
GET /getusers/1
POST /users/1/delete
POST /users/1/create
```

正确的用法为：

```
GET /users/1
DELETE /users/1
PUT /users/1
```

关注请求头

一定要看请求头信息，并给予正确的状态码。举个例子，假设服务器端只能返回 JSON 格式，如果客户端的头信息的 Accept 字段要求返回 application/xml，这个时候就不应该返回 application/json 类型的数据，而应该返回 406 错误。

合理使用请求方法和状态码

不能一味使用 GET 和 POST，返回 200，不合理的请求方法可能让未来的维护者或者合作方感到迷惑。关于方法语义的说明，可参考表 5.1。

表 5.1 方法语义的说明

方 法	语 义
OPTIONS	用于获取资源支持的所有 HTTP 方法
HEAD	用于只获取请求某个资源返回的头信息
GET	用于从服务器获取某个资源的信息： 1. 完成请求后，返回状态码 200 OK 2. 完成请求后，需要返回被请求的资源详细信息
POST	用于创建新资源： 1. 创建完成后，返回状态码 201 Created 2. 完成请求后，需要返回被创建的资源详细信息
PUT	用于完整的替换资源或者创建指定身份的资源，比如创建 id 为 123 的某个资源： 1. 如果是创建了资源，则返回 201 Created 2. 如果是替换了资源，则返回 200 OK
PATCH	用于局部更新资源： 1. 完成请求后，返回状态码 200 OK 2. 完成请求后，需要返回被修改的资源详细信息 3. 完成请求后，需要返回被修改的资源详细信息
DELETE	用于删除某个资源，完成请求后返回状态码 204 No Content

正确地使用 REST

REST 服务器是无状态的。在有分页的时候，它并不能知道你当前访问到了什么位置，前一页和后一页的地址是什么，这个关系需要客户端来维护。但是“下一页资源”这样的业务逻辑是需要服务端来提供的。例如，下面例子的返回是不完整的：

```
Status: 200 OK
[
  {
    "id": 1,
    "url": "https://api.dongwm.com/users/1/",
  },
  {
    "id": 2,
    "url": "https://api.dongwm.com/users/2/",
  }
]
```

返回的结果没有告诉我们是否有下一页，也没有告诉我们符合条件的记录总数。可以添加 Link 和 X-Total-Count 头来提供这样的功能：

```
Status: 200 OK
X-Total-Count: 210
Link: <https://api.dongwm.com/users?page=2>; rel="next",
      <https://api.dongwm.com/users?page=5>; rel="last"
[
  {
    "id": 1,
    "url": "https://api.dongwm.com/users/1/",
  },
  {
    "id": 2,
    "url": "https://api.dongwm.com/users/2/",
  }
]
```

rel 的值还可以是 first、self 和 prev，客户端只需要根据 Link 中提供的链接就可以找到全部的符合条件的条目。

还有两处容易出错的地方需要注意：

- 使用“201 Created”响应时，应该带 Location，指向新建资源的地址。
- 使用“405 Method Not Allowed”响应时，应该带有 Allow 头，告诉客户端对该资源有效的 HTTP 方法。

对输出的结果不再包装

body 中应该直接放数据，不要多层封装。下面有个不恰当的响应的例子：

```
HTTP/1.1 200 OK
{
  'success': true,
  'data': {'id':1, 'name': 'xiaoming'},
}
```

直接返回 data 中的数据就好了：

```
HTTP/1.1 200 OK
{'id':1, 'name': 'xiaoming'}
```

因为通过状态码“200 OK”就可以知道结果是正确的，也没有必要添加“success”字段。



如果 API 使用者确实由于某种原因无法访问返回头，或者 API 需要支持交叉域请求（例如通过 jsonp），这两种情况下还是需要包装的。

不要做出错误的提示

当访问出错或者响应的结果不符合预期时，不应该返回 200 作为状态码。哪怕返回的结果中也包含了错误原因，因为在没有充分的文档说明前提下，客户端可能会缓存成功的 HTTP 请求。

使用嵌套对象序列化

对象应该合理地嵌套，不应该都在一个层次上。如下的格式是不正确的：

```
{
  'id': 1,
  'post_id': '10001',
  'post_name': 'Post1',
  'post_content': 'this is a post'
}
```

尽可能把相关联的资源信息内联在一起。应该把 post 作为一个键：

```
{
  'id': 1,
  'post': {
    'id': '10001',
```

```
        'name': 'Post1',
        'content': 'this is a post'
    }
}
```

版本

常见的区分版本的方法有三种：

- 保存在 URI 中。比如 “https://api.dongwm.com/api/v2”。
- 放在请求头中。比如 GitHub 的用法：“Accept: application/vnd.github.v3+json”。
- 自定义请求头。比如，“X-Api-Version: 1”。

第三种方式不推荐，推荐使用第一种。

URI 失效和迁移

随着业务发展，会出现一些 API 失效或者迁移。对失效的 API，应该返回 “404 not found” 或 “410 gone”；对迁移的 API，返回 301 重定向。

信息过滤

URL 通常最好越简短越好，对结果过滤、排序和搜索相关的功能都应该通过参数实现。一些常见的参数用法如表 5.2 所示。

表 5.2 常见的参数及其用法

参 数	含 义
offset=0&limit=10	指定返回记录的数量，offset 也可以用 start 这个名字
offset=10	指定返回记录的开始位置
page=2&per_page=100	指定第几页，以及每页的记录数
sortby=name&order=asc	指定返回结果按照哪个属性排序，以及排序的顺序
sort=age,desc	多个排序条件组合

速度限制

为了避免请求泛滥，给 API 设置速度限制很重要。为此，RFC 6585 (<https://tools.ietf.org/html/rfc6585>) 引入了 HTTP 状态码 429 (too many requests)。加入速度限制功能之后，应该提示用户。可以参照 GitHub 的返回头，如下所述。

- X-RateLimit-Limit: 当前时间段允许的并发请求数。
- X-RateLimit-Remaining: 当前时间段保留的请求数。
- X-RateLimit-Reset: 当前时间段剩余的秒数。

我们看一个真实的例子:

```
> curl -i https://api.github.com/users/whatever
HTTP/1.1 200 OK
Date: Mon, 01 Jul 2013 17:27:06 GMT
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 56
X-RateLimit-Reset: 1372700873
```

缓存

数据内容在一段时间不会变动，这个时候我们就可以合理地减少 HTTP 响应内容。应该在响应头中携带 Last-Modified、ETag、Vary、Date 等信息，客户端可以在随后请求这些资源时，在请求头中使用 If-Modified-Since、If-None-Match 等来确认资源是否经过修改。如果资源没有做过修改，那么就可以响应“304 Not Modified”，并且不在响应实体中返回任何内容。

我们看看 GitHub 的用法（隐藏了无关的自定义头）：

```
> http https://api.github.com/users/dongweiming --headers
HTTP/1.1 200 OK
Cache-Control: public, max-age=60, s-maxage=60
Content-Encoding: gzip
Content-Security-Policy: default-src 'none'
Content-Type: application/json; charset=utf-8
Date: Thu, 04 Feb 2016 14:05:03 GMT
ETag: W/"393f2b88fc9073927d2dda6f43318c8a"
Last-Modified: Sat, 16 Jan 2016 09:14:15 GMT
Server: GitHub.com
Status: 200 OK
Transfer-Encoding: chunked
Vary: Accept
Vary: Accept-Encoding
```

我们可以通过 If-Modified-Since 实现缓存:

```
> http https://api.github.com/users/dongweiming 'If-Modified-Since: Thu, 04 Feb 2016
14:05:03 GMT' --headers
HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=60
Content-Security-Policy: default-src 'none'
Date: Thu, 04 Feb 2016 14:06:43 GMT
Last-Modified: Sat, 16 Jan 2016 09:14:15 GMT
Server: GitHub.com
Status: 304 Not Modified
Vary: Accept-Encoding
```

当生成请求的时候, 在 HTTP 头里面加入 ETag。其中包含请求的校验和与哈希值, 这个值在输入变化的时候也应该变化。如果输入的 HTTP 请求包含 If-None-Match 头以及一个 ETag 值, 那么 API 应该返回 “304 Not Modified” 状态码, 而不是常规的输出结果:

```
> http https://api.github.com/users/dongweiming 'If-None-Match: "393
f2b88fc9073927d2dda6f43318c8a"' --headers
HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=60
Content-Security-Policy: default-src 'none'
Date: Thu, 04 Feb 2016 14:07:17 GMT
ETag: "393f2b88fc9073927d2dda6f43318c8a"
Last-Modified: Sat, 16 Jan 2016 09:14:15 GMT
Server: GitHub.com
Status: 304 Not Modified
Vary: Accept-Encoding
```

并发控制

缺少并发控制的 PUT 和 PATCH 请求可能导致 “更新丢失”。这个时候可以使用 Last-Modified 和 ETag 头来实现条件请求。具体原则如下:

- 客户端发起的请求如果没有包含 If-Unmodified-Since 或者 If-Match 头, 就返回状态码 “403 Forbidden”, 在响应正文中解释为何返回该状态码。
- 客户端发起的请求所提供的 If-Unmodified-Since 或者 If-Match 头与服务器记录的实际修改时间或 ETag 值不匹配时, 返回状态码 “412 Precondition Failed”。
- 客户端发起的请求所提供的 If-Unmodified-Since 或者 If-Match 头与服务器记录的实际修改时间或 ETag 的历史值匹配, 但资源已经被修改过时, 返回状态码 “409 Conflict”。

- 客户端发起的请求所提供的条件符合实际值，就更新资源，响应“200 OK”或者“204 No Content”，并且包含更新过的 Last-Modified 和/或 ETag 头，同时包含 Content-Location 头，其值为更新后的资源 URI。

本节参考了开源项目“HTTP 接口设计指北”(<http://bit.ly/2azgIBP>)，如果想了解更多内容，请阅读此项目。

使用 Ajax

之前看到的例子都是整页地提交，如果提交失败就重新刷新页面，显示错误信息；如果提交正确，就返回完整的新页面或者跳转到其他页面。重新刷新页面和返回新页面意味着整个页面的内容都要重新加载，但事实上其中大部分 HTML 是相同的。这种传统方法浪费了很多带宽，加载得更慢，用户体验不好。另外一个问题是，刷新的页面里应该包含之前未提交成功的数据，不应该让用户重复输入，这也需要大量的逻辑来保证。

大型网站的首页内容都很丰富，比如豆瓣的匿名（未登录状态）首页，除了登录表单，还有顶部导航栏、热点内容和各产品线的推荐内容等。登录时，可不可以在不刷新页面的前提下，只向服务器发送和取回登录必要的数据呢？

Ajax 是 Asynchronous JavaScript and XML 的简称，通过 Ajax 向服务器发送 HTTP 请求，接收服务器返回的 JSON 数据，然后使用 JavaScript 修改网页的局部来实现更新和提交。现在绝大多数 Web 应用都在使用 Ajax。

jQuery 是一个高效、精简并且功能丰富的 JavaScript 工具库，它提供的 API 易于使用且兼容全部主流浏览器，如果不是专业的前端工程师，使用它也可以很方便地操作文档对象、选择文档对象模型（DOM）元素、处理事件、创建动画效果、开发 Ajax 程序等。

首先下载 jQuery：

```
> wget https://code.jquery.com/jquery-2.2.4.min.js -O static/javascripts/jquery.min.js
```



jQuery 2.x 与其 1.x 的 API 一样，但是前者只支持 IE 9 及以上的浏览器版本。

登录页面（signin.html）如下：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Signin</title>
    <script src="{{ url_for('static', filename='javascripts/jquery.min.js') }}"></script>
```

```
<style>
    #result {
        margin-top: 20px;
        color: red;
    }
</style>
</head>

<body>
    <div class="container">
        <form action="/signin" method="post" role="form">
            <h2>Please Sign In</h2>
            <input type="text" name="username" placeholder="Username" required
                autofocus>
            <input type="password" name="password" placeholder="Password" required>

            <button class="btn" type="button">SignIn</button>
        </form>

        <div id="result"></div>
    </div>

    <script type='text/javascript'>
        $(function() {
            $('.btn').click(function() {
                var $result = $('#result');
                var $username = $('input[name="username"]').val();
                var $password = $('input[name="password"]').val();
                $.ajax({
                    url: '/signin',
                    data: $('form').serialize(),
                    type: 'POST',
                    dataType: 'json'
                }).done(function(data) {
                    if (!data.r) {
                        $result.html(data.rs);
                    } else {
                        $result.html(data.error);
                    }
                });
            });
        });
    </script>
</body>
</html>
```

这个页面主要包含如下部分。

1. **head**: head 标签中可以引用 JavaScript 脚本和样式表 (CSS)、提供元信息等。jquery.min.js 这种全局的工具库应该放在最上面, 在渲染页面时 jQuery 可以更早加载。除此之外, style 标签还给 id 为 result 的 div 元素添加了样式。
2. **form**: 就是登录表单, 其中 action 是提交时请求的地址, method 表示提交方法。表单中有两个 input 标签, 分别为用户名和密码, 还有一个用来提交的按钮。
3. **script**: 最下面的 script 标签中使用 jQuery 监控类名为 btn 的元素的点击事件。单击“提交”按钮则获取表单中的用户名和密码, 通过 \$.ajax 函数发送 POST 请求。其中 done 是执行成功的回调函数, 返回的数据叫作 data, 如果 data.r 为 0, 表示登录成功, 给 id 为 result 的 div 添加 data.rs 中的内容, 否则添加 data.error 中的内容。由于 script 会获取页面元素, 应该在页面元素之后出现, 所以一般都在 body 的最下面。

上述代码中有一些使用 jQuery 的细节。

- “\$(function() { ... })” 将会在浏览器加载完页面的基础内容之后立即执行。
- jQuery 可以使用标签和 CSS 选择器来选取 HTML 元素。举几个例子来看一看:

\$("#p"): 选取 <p> 元素。

\$("#p.intro"): 选取所有 class="intro" 的 <p> 元素。

\$("#p#demo"): 选取所有 id="demo" 的 <p> 元素。

\$("#input[name='password']"): 选取所有 name="password" 的 <input> 元素。

- 上例使用了 “\$result.html” 设置内容, html() 可以设置或返回所选元素的内容 (包括 HTML 标记)。除此之外, 还有 text() (设置或返回所选元素的文本内容) 和 val() (设置或返回表单字段的值)。使用 html() 更灵活, 因为可以在返回的结果中添加标签和样式。

看一看页面视图:

```
@app.route('/')
def index():
    return render_template('chapter5/section3/signin.html')
```

```
@app.route('/signin', methods=['POST'])
def signin():
    username = request.form['username']
    password = request.form['password']
    error = None
    if len(username) < 5:
        error = 'Password must be at least 5 characters'
    if len(password) < 6:
        error = 'Password must be at least 8 characters'
```

```
elif not any(c.isupper() for c in password):
    error = 'Your password needs at least 1 capital'
if error is not None:
    return jsonify({'r': 1, 'error': error})
return jsonify({'r': 0, 'rs': '0k'})
```

视图分为两个：

- 首页接受 GET 请求，渲染 `signin.html`。
- `/signin` 页面即上面 Ajax 设置的请求地址，接收用户名和密码，处理之后用 JSON 格式返回结果。需要强调的是，不仅后端需要验证用户名和密码，前端也应该通过 JavaScript 验证用户名和密码的格式是否符合，如果不符合，则直接提示错误而不用产生一次请求，减少后端压力。

使用 `$.ajax` 的时候，提交的数据使用了 “`$(‘form’).serialize()`”，它通过序列列表单值，创建 URL 编码的文本字符串。序列化的值可在生成 Ajax 请求时用在 URL 查询字符串中；否则需要挨个字段拼出来：

```
data: {'username': $username, 'password': $password}
```

使用 fetch 实现 Ajax

传统 Ajax 指的是 XMLHttpRequest (XHR)，JavaScript 通过它来执行异步请求，jQuery 帮我们把 XMLHttpRequest 封装起来，成为 `$.ajax`、`$.get` 和 `$.post` 等函数。XHR 在设计上不符合职责分离原则，输入、输出以及状态都放在同一对象中，而且受 XML 影响，其命名也很复杂。

在 ECMAScript 2015 (ES6) 规范中有一个新的特性：Promise。Promise 对象用于延迟 (deferred) 计算和异步 (asynchronous) 计算，一个 Promise 对象代表着一个还未完成，但预期将来会完成的操作。Fetch (<https://github.com/github/fetch>) API 就是基于 Promise 设计的。举个 Ajax 返回 JSON 数据的例子，如果是用原生 XHR 写，会是这样：

```
var data = new FormData();
data.append('username', $username);
data.append('password', $password);

var xhr = new XMLHttpRequest();
xhr.open('POST', '/signin');

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var data = JSON.parse(xhr.responseText);
        if (!data.r) {
            $result.html(data.rs);
        }
    }
}
```

```
        } else {
            $result.html(data.error);
        }
    };
};

xhr.send(data);
```

如果使用 fetch 则可以写成这样:

```
fetch('/signin', {
    method: 'POST',
    body: data
}).then(function(response) {
    return response.json();
}).then(function(data) {
    if (!data.r) {
        $result.html(data.rs);
    } else {
        $result.html(data.error);
    }
});
```

从 Firefox 39 以及 Chrome 42 开始, 它们已经支持了 Fetch API。为了保证其他浏览器或者低版本的 Firefox 以及 Chrome 也能使用, 需要使用 GitHub 开源的 polyfill 脚本:

➤ `wget https://github.com/github/fetch/raw/master/fetch.js -O static/javascripts/fetch.js`

在模板中加载这个脚本:

```
<script src="{{ url_for('static', filename='javascripts/fetch.js') }}"></script>
```

如果再使用 ES 7 提供的 `async/await` API, 就可以像写同步代码一样写异步代码了:

```
try {
    let response = await fetch('/signin', {
        method: 'POST',
        body: data
    })
    let data = await response.json();
    if (!data.r) {
        $result.html(data.rs);
    } else {
        $result.html(data.error);
    }
} catch(e) {
    console.log("Oops, error", e);
}
```

第 6 章

网站架构

本章将介绍网站架构中的一些主流组件以及豆瓣的基础架构，主要包含如下内容：

- 了解 WSGI 协议。
- 主流的 Python 应用服务器的特点和适用方法。
- 使用 Nginx 和 Python 应用服务器部署 Flask 应用。
- 介绍豆瓣开源的 Libmc 和豆瓣常用的缓存使用方式。
- 举例说明 Redis 的几个应用场景，包含使用 MessagePack 进行序列化和反序列化工作。
- 介绍使用 NoSQL 的原因和场景。
- 使用 pymongo，并用 Mongoengine 重构文件托管服务的模型。
- MongoDB 索引、高可用和分片的经验。
- 以豆瓣的基础架构为原型，展示主流大型网站的架构模式，并详细介绍相关重要模式，以及 Web 前端的性能优化经验。

Python 应用服务器

Python 的 Web 框架（如 Flask、Django）自带 Web 服务器的目的是用于开发，而不是生产环境。那么，生产环境中有哪些应用服务器可以选择呢？

我们首先了解一下 WSGI 协议。

WSGI 协议

Web 框架和 Web 服务器之间需要进行通信，如果在设计时它们之间无法相互匹配，那么对框架的选择就会限制对 Web 服务器的选择，这显然是不合理的。这时候需要设计一套双方都遵守的接口。WSGI 是 Python Web Server Gateway Interface 的简称。WSGI 标准在 PEP 333 中定义并被许多框架实现，它规定了一种在 Web 服务器与 Web 应用程序/框架之间推荐的标准接口，以确保 Web 应用程序在不同的 Web 服务器之间具有可移植性。在后来的 PEP 3333 中添加了 Python 3 的支持和更多相关的说明。有了通用的 WSGI 协议，Web 开发者就能够任意选择适合自己的组合，而 Web 服务器和 Web 框架的开发者们也能够把精力集中到各自的领域。

常见的 WSGI 容器

WSGI 是一个同步接口，所以 Tornado 的 WSGI 容器是无法实现异步的。主流的选择是 Gunicorn 和 uWSGI。

Gunicorn

Gunicorn 易于配置，兼容性好，CPU 消耗很少，在豆瓣使用广泛。它支持多种 Worker 模式，推荐的模式有如下几种。

- 同步 Worker：默认模式，也就是一次只处理一个请求。
- 异步 Worker：通过 Eventlet、Gevent 实现的异步模式。
- 异步 IO Worker：目前支持 gthread 和 gaiohttp 两种类型。

我们先安装它：

```
> pip install gunicorn
```

Gunicorn 的启动非常简单，语法如下：

```
> gunicorn [OPTIONS] MODULE_NAME:VARIABLE_NAME
```

我们看一个最简单的例子（app.py）：

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route('/')
def hello_world():
```

```
    return 'Hello'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9000)
```

可以使用如下命令启用应用：

```
> gunicorn --workers=3 chapter6.section1.app:app -b 0.0.0.0:9000
```

chapter6.section1 是模块目录的名字，第一个 app 是模块文件名字，第二个 app 是文件中 Flask 实例的名字。

Worker 的数量并不是越多越好，推荐值是 CPU 的个数 $\times 2 + 1$ ，CPU 个数使用如下的方式获取：

```
> python -c 'import multiprocessing; print multiprocessing.cpu_count()'
1
```

虚拟机只有 1 个 CPU，可以启动 3 个 Worker ($1 \times 2 + 1$)。

uWSGI

uWSGI 是使用 C 编写的，实现了自有的 uwsgi 协议的 Web 服务器。它自带丰富的组件，其中核心组件包含进程管理、监控、IPC 等功能，实现应用服务器接口的请求插件支持多种语言 and 平台，比如 WSGI、Rack、Lua WSAPI，网关组件实现了负载均衡、代理和路由功能。

我们先安装它：

```
> pip install uwsgi
```

它的启动命令要相对复杂一些，有非常多的选项：

```
> uwsgi --http 0.0.0.0:9000 --wsgi-file chapter6/section1/app.py --callable app --
    processes 4 --threads 2 --stats 0.0.0.0:5000
```

上面的命令表示启动了 4 个进程，每个进程使用 2 个线程，而且开启了 5000 的 Web 接口，返回监控 uWSGI 的信息，以及不同进程和线程的详细使用情况。

使用 uWSGI 时，一定要先读一读官网的 *Things to know (best practices and “issues”)* READ IT !!! (<http://bit.ly/2bd1ULz>)。其中有两点非常重要：

- --http-socket 和 --http 其实是完全不同的两个选项。如果想直接裸跑 uWSGI，应该使用 --http，它产生一个额外的进程将请求转发给 Workers，如果希望它被反向代理（比如和 Nginx 一起来用），应该使用 --http-socket。

- 合理的进程数和线程数不能简单地通过 $\text{CPU} * 2$ 来计算得出，需要不断尝试而找到最佳值。除了 uWSGI 提供的监控服务器，uwsgitop (<http://bit.ly/2bo5z7s>) 也是一个非常好的帮助我们找到这个最佳值的工具。uwsgitop 接收的唯一参数就是对应的 --stats 的参数（比如上面的 0.0.0.0:5000）或者 socket 文件的地址。

除此之外，Twisted Web 和 Gevent 也提供了 WSGI 容器，但不推荐在生产环境中使用。

Web 服务器 Nginx

用 Python 语言开发的站点使用的 Web 服务器主要有 Nginx、Apache 以及淘宝的 Tengine。它们和之前说的应用服务器有什么区别呢？

Web 服务器与应用服务器的区别

1. Web 服务器负责处理 HTTP 协议，应用服务器既可以处理 HTTP 内容，也能处理其他协议，比如 RPC。
2. Web 服务器用于处理静态页面的内容，对于脚本语言（比如 Python）产生的动态内容，它通过 WSGI 接口交给应用服务器来处理。
3. 一般应用服务器都集成了 Web 服务器，自带的应用服务器甚至可以支持应用级别的功能，比如连接池、事务支持、消息服务等。虽然集成了 Web 服务器，主要是为了调试方便，出于性能和稳定性考虑，应用服务器并不能在生产环境中使用。

为什么要选择 Nginx

Nginx 是由 Igor Sysoev 在 2004 年发布的一个开源、高性能的 HTTP 服务器和反向代理，它还可以用来作为 IMAP/POP3 的代理服务器。它被广泛使用，有如下原因：

- 作为 Web 服务器，它处理静态文件、索引文件的效率非常高。
- Nginx 的设计非常注重效率，它支持 epoll/kqueue 等网络 I/O 模型，可以最大支持 5 万个并发链接，并且只占用很少的内存资源。
- 稳定性高，宕机的概率很低。
- 强大的反向代理和负载均衡功能，平衡集群中各个服务器的负载压力。
- 配置简洁。配置文件通俗易懂，上手很容易。
- 支持热部署，可以在不间断服务器的情况下对软件进行升级。

- 提供健康检查支持，当后端出现问题时，就不再往这个后端分发请求，并且还会做后续的检查，直到这个后端恢复正常。

安装 Nginx

软件包安装方法主要有两种，使用系统提供的二进制包安装，或源码安装。大部分场景下使用系统默认包就够了，但对于如下情况应该使用源码安装：

- 对软件的精简度、性能有非常高的要求。系统自带软件包中的选项和参数都中规中矩，有些设置在复杂的生产环境中并不适用。源码安装自由度很高，可以根据需要在编译的时候开启/关闭某些属性。
- 对软件打过补丁（Patch）。
- 源码安装提供了统一的安装方式，软件可以被应用到多种平台中。

通过源码安装，如果要定制，就需要在第一步完成。建议参考对应平台的默认包的编译参数。笔者通常也会参考 Gentoo 的配置选项。可以到 www.servers/nginx 包页面（<http://bit.ly/2b4Y8jJ>）找到对应版本的 ebuild 文件，这里可以参照 nginx-1.9.10-r3.ebuild（<http://bit.ly/2b4ZJ9a>）。

当然，你也可以自己构建系统包，比如在 Ubuntu 下可以自己构建维护 deb 包。

本书都使用系统提供的安装方式，通过 apt-get 安装 Nginx：

```
> sudo apt-get install nginx -yq
```

安装完之后 Nginx 已经启动了。

使用 Nginx 部署 Flask 应用

部署 Flask 应用时，通常都是使用一种 WSGI 应用服务器搭配 Nginx 作为反向代理。什么是反向代理呢？

反向代理和正向代理

正向代理，作为一个媒介将互联网上获取的资源返回给相关联的客户端。代理和客户端在一个局域网，对于服务端是透明的。反向代理，根据客户端的请求，从后端的服务器上获取资源，然后再将这些资源返回给客户端。代理和服务器在一个局域网，对客户端是透明的。

Nginx 是反向代理的最佳选择，那么为什么需要反向代理呢？反向代理有如下作用：

- 提高动态语言的 I/O 处理能力，Python、PHP、Java 这样的动态服务的 I/O 处理能力不高，反向代理可以缓冲请求，交给后端一个完整的 HTTP 请求，同样，Nginx 也可以缓冲响应，也达到了减轻后端的压力。
- 加密和 SSL 加速。
- 安全。它保护和隐藏了原始资源服务器，还可以用作应用防火墙防御一些网络攻击，比如 DDoS。
- 负载均衡。它帮应用服务器分配请求，以达到资源使用率最佳、吞吐率最大、响应时间最小的目的。
- 缓存静态内容。代理缓存通常可以满足相当数量的网站请求，大大降低应用服务器上的负载。
- 支持压缩。通过压缩优化可以提高网站访问速度，还能大大减少带宽的消耗。

Nginx 配置

Nginx 的配置文件是以块（block）的形式组织的。每个块以一个花括号（{}）来表示，主要有 6 种块，如表 6.1 所示。

表 6.1 6 种配置文件块及其含义

块	含 义
main	全局设置，包含一些 Nginx 的基本控制功能。它在配置的顶层，之下包含 events 和 http 这两种块
events	事件设置，控制 Nginx 处理连接的方式
http	HTTP 设置，在它之下包含 server 和 upstream 这两种块
server	主机设置
upstream	负载均衡设置
location	URL 模式设置，在 server 层之下。server 可以包含多个 location 块

我们来看一个适用于 Nginx+Gunicorn 模式的 Nginx 配置（nginx_gunicorn.conf），由于篇幅限制，只列出和 Python 应用服务器相关的部分：

```
http {
    upstream frontends {
        server 127.0.0.1:8000;
        server 127.0.0.1:8001;
```

```

server 127.0.0.1:8002;
# server unix:/tmp/gunicorn.sock; # 使用UNIX Socket文件的方式
}

server {
    listen 80;
    server_name localhost;
    location ^~ /static/ {
        root /home/ubuntu/web_develop/static;
    }
}

# 通过下面的方式把这些类型文件缓存30天, 前提是保证这些文件是不经常改变的
location ~* \.(woff|eot|ttf|svg|mp4|webm|jpg|jpeg|png|gif|ico|css|js)$ {
    expires 30d;
}

location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Scheme $scheme;
    proxy_redirect off;
    proxy_pass http://frontends;
}
}
}

```

替换默认的 Nginx 配置:

```

> sudo cp chapter6/section2/nginx_gunicorn.conf /etc/nginx/nginx.conf
> sudo /etc/init.d/nginx reload

```

在重启之前, 可以使用 configtest 用来验证配置文件语法的正确性:

```

> sudo /etc/init.d/nginx configtest
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful

```

负载均衡算法

Nginx 的负载均衡模块目前支持如下 4 种调度算法。

1. round-robin: Nginx 默认的轮询算法, 每个请求按时间顺序逐一分配到不同的后端服务器, 如果后端某台服务器宕机, 故障系统将被自动剔除, 使用户访问不受影响。

可以通过 `weight` 指定轮询权值, `weight` 值越大, 该服务器被访问的概率越高, 这主要用于后端每个服务器性能不均的情况。

2. `least_conn`: 请求会被发送到活跃连接数最少的服务器上。配置例子如下:

```
upstream backend {
    least_conn; # 默认的轮询算法不需要指定, 如选择其他算法需要指定算法类型
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
}
```

3. `ip_hash`: 按访问 IP 的哈希结果分配请求。也就是说, 来自同一个 IP 的用户会固定访问一个后端服务器。

```
upstream backend {
    ip_hash;
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
}
```

4. `hash`: 按某个键的哈希结果分配 (键可以是文本、变量等) 请求。

```
upstream backend {
    hash $request_uri; # 根据请求地址生成哈希结果
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
}
```

负载均衡支持的状态参数

Upstream 模块中支持 4 种状态类型, 如表 6.2 所示。

表 6.2 Upstream 模块支持的 4 种状态类型

状态参数	含 义
<code>down</code>	当前的服务器暂时不参与负载均衡, 对这个服务器的请求会自动发送到下一个服务器
<code>max_fails</code>	允许请求失败次数, 默认为 1。当超过最大次数时, 返回对应的 <code>XX_next_upstream</code> 模块定义的错误 (XX 可能是 <code>proxy</code> 、 <code>fastcgi</code> 、 <code>uwsgi</code> 、 <code>scgi</code> 、 <code>memcached</code> 等)
<code>fail_timeout</code>	在经历了 <code>max_fails</code> 次失败后, 暂停服务的时间, 默认是 10 s。 <code>max_fails</code> 可以和 <code>fail_timeout</code> 一起使用
<code>backup</code>	预留的备份服务器。当其他所有的非 <code>backup</code> 服务器出现故障或者忙的时候, 才会请求 <code>backup</code> 的服务器

下面来展示一个复杂的负载均衡配置:

```
upstream backend {  
    server 127.0.0.1:8000 down;  
    server 127.0.0.1:8002 max_fails=3 fail_timeout=30;  
    server 127.0.0.1:8003 weight=5 fail_timeout=2;  
    server 127.0.0.1:8004 backup;  
}
```

通过 Gunicorn 启动 Flask 应用

Nginx 负载均衡设置了 8000~8002 这三个后端端口, 为了演示, 这里只启动 8000 端口的 Gunicorn 实例:

```
> gunicorn -w 3 chapter6.section1.app:app -b :9000
```

现在可以通过 Nginx 访问 Flask 应用了:

```
> http http://localhost  
HTTP/1.1 200 OK  
Connection: keep-alive  
Content-Length: 5  
Content-Type: text/html; charset=utf-8  
Date: Wed, 01 Jun 2016 15:53:58 GMT  
Keep-Alive: timeout=20  
Server: nginx/1.10.0 (Ubuntu)
```

Hello

通过 uWSGI 启动 Flask 应用

使用 uWSGI 的 Nginx 配置和上面的相差不多, 这里只列出关键的 http 部分 (nginx_uwsgi.conf):

```
http {  
    upstream frontends {  
        server unix:/tmp/uwsgi.sock; # 选择 Unix Socket 的方式  
        server 127.0.0.1:8000; # 可以和 TCP/IP Socket 方式混用  
    }  
  
    server {  
        ...  
        location / {  
            uwsgi_pass frontends;  
            include uwsgi_params;  
            ...  
        }  
    }  
}
```

```
    }  
}  
}
```

由于选择了 UNIX Socket 方式，所以启动 uWSGI 的方式略有不同：

```
> uwsgi --socket /tmp/uwsgi.sock --wsgi-file chapter6/section1/run.py --callable app --  
processes 1 --threads 1
```

TCP/IP Socket 和 UNIX Socket 区别

UNIX Socket 是同一台服务器上不同进程间的通信机制。TCP/IP Socket 是网络上不同服务器之间进程的通信机制，也可以让同一服务器的不同进程通信。

Postgres 的一位核心开发者曾经做过实验（<http://bit.ly/28UN7nG>），证明 UNIX Socket 的方式比 TCP/IP Socket 方式要快 31%，FreeBSD 上也曾经有过讨论（<http://bit.ly/28Qx1Jj>）。所以，在同一个服务器上应该优先选择 UNIX Socket 方式。

缓存系统 Memcached

绝大多数的 Web 应用都把数据存在数据库里面，应用服务器从中读取数据。随着数据量和访问量的增加，频繁通过直接访问数据库获得数据的方式对数据库造成很大的负担。Memcached 是一个高性能的分布式内存对象缓存系统，用在动态 Web 应用中减轻数据库负载。它通过在内存中缓存数据来减少读取数据库的次数，从而提高 Web 应用的速度。

Memcached 的守护进程使用 C 编写，客户端则可以使用任何语言，通过 Memcached 协议与守护进程通信。常见的 Python 客户端有如下几种。

1. Python-memcached：纯 Python 实现的客户端，但是只实现了 Memcached 的基本操作功能。
2. Pymemcache：Pinterest 开源的纯 Python 实现的客户端，比 Python-memcached 要快。
3. Python-Libmemcached：豆瓣开源的使用 Cython 实现的 Libmemcached 的客户端。Libmemcached 是 Memcached 的 C/C++ 客户端，它支持一致性哈希、分布式、同步/异步传输等功能。豆瓣用的 Libmemcached 是打过补丁的。
4. Pylibmc：它也是 Libmemcached 的一个使用 C 编写的 Python 封装的客户端。
5. Libmc（<http://bit.ly/28SHgfG>）：它是使用 C++ 和 Cython 编写的一个轻量高效的 Memcached Python 客户端，支持以一致性哈希环的方式一次性与多个 Memcached 节点交互。Libmc 使用了类似 Pylibmc（<http://bit.ly/28W6bn7>）的基准测试方法（<http://bit.ly/28W6bn7>）。

//bit.ly/28S8wuh), 对比了 Pylibmc、Python-memcached 和 Libmc 的性能, 测试结果可以在豆瓣的 Travis-CI 集成 (<http://bit.ly/28SkeVh>) 中找到。可以看到在所有的测试中都是 Libmc 最快。豆瓣在生产环境中已经完全使用 Libmc 替换了 Libmemcached 加 Python-Libmemcached 的组合。原因是 Libmemcached 代码非常复杂且 bug 多, 豆瓣所使用的分支也与上游脱节, 不可维护。

Libmc 安装配置

首先安装 Memcached:

```
> sudo apt-get install memcached
```

安装之后 Memcached 已经自动启动了:

```
> ps -ef |grep memcached
memcache 3108      1  0 16:22 ?           00:00:00 /usr/bin/memcached -m 64 -p 11211 -u
memcache -l 127.0.0.1
```

这是默认的启动方式, 表示使用 memcache 这个用户, 最大占用 64 MB 内存, 监听本机的回路地址和 11211 端口。默认只使用 64 MB 内存实在太小了, 在生产环境中需要根据实际内存情况使用更大的值。下面会使用 Memcached 分布式处理缓存, 所以再启动两个监听其他端口的实例:

```
> /usr/bin/memcached -m 64 -p 11212 -u memcache -l 127.0.0.1 -d
> /usr/bin/memcached -m 64 -p 11213 -u memcache -l 127.0.0.1 -d
```

接着安装 Libmc。如果使用 Vagrant 的方式, 编译需要的内存将超出虚拟机所能提供的大小, 需要创建 swap 分区用户替代内存资源:

```
> sudo dd if=/dev/zero of=/swapfile bs=64M count=48
> sudo mkswap /swapfile
> sudo swapon /swapfile
```

这样, 我们就可以借用 3 GB 硬盘作为内存来使用了。Docker 方式不需要使用 swap 分区。

```
> pip install libmc
```

我们使用 Libmc 的配置如下 (mc.py):

```
from libmc import (
    Client, MC_HASH_MD5, MC_POLL_TIMEOUT, MC_CONNECT_TIMEOUT, MC_RETRY_TIMEOUT
)

mc = Client(
    [
```



```
        'localhost',  
        'localhost:11212',  
        'localhost:11213 mc_213'  
    ],  
    do_split=True,  
    comp_threshold=0,  
    noreply=False,  
    prefix=None,  
    hash_fn=MC_HASH_MD5,  
    failover=False  
)  
  
mc.config(MC_POLL_TIMEOUT, 100)  
mc.config(MC_CONNECT_TIMEOUT, 300)  
mc.config(MC_RETRY_TIMEOUT, 5)
```

我们解析下这个获得 mc 实例的程序。

- libmc.Client 接受的第一个参数就是 Memcached 服务器的列表，格式是“host-name[:port] [alias]”。其中端口和别名是可选项，若未指定端口，则默认端口为 11211，如 localhost:11211 和 localhost 是等价的。
- do_split：默认值为 False，会拒绝大于 1 MB 的值的存储；如果设置为 True，小于 10 MB 的值会被切分成多个块，但是不能存储大于 10 MB 的值。
- comp_threshold：所有类型的值都会被编码为字符串，如果设置的这个阈值等于 0，字符串不会使用 zlib 压缩；如果这个字符串的长度大于设置的阈值且不为 0，就会使用 zlib 压缩。默认值是 0。
- noreply：表示是否开启 noreply 模式。在这种 noreply 模式下，更新缓存的 set 操作可以不需要 Memcached 服务端响应，这使得 set 操作非常快。默认值为 False。
- prefix：缓存键的前缀，常用于区分不同的环境中相同的缓存键。
- hash_fn：对键做哈希的函数标识，可选值有 MC_HASH_MD5、MC_HASH_FNV1_32、MC_HASH_FNV1A_32 和 MC_HASH_CRC_32。默认值是 MC_HASH_MD5。
- failover：表示当前服务器不可用时，是否做故障转移到写一个服务器。默认值为 False。
- MC_POLL_TIMEOUT：执行 Memcached 的 set/get 操作的超时时间，单位为 ms。
- MC_CONNECT_TIMEOUT：连接 Memcached 的超时时间，单位为 ms。
- MC_RETRY_TIMEOUT：当 Memcached 服务器不可用等情况下，重试链接的时间，单位为 s。这种重试一直持续到服务恢复。

使用原生 SQL 缓存

现在参考豆瓣缓存服务客户端 `douban-mc` (<http://bit.ly/28Y4bsO>) 实现一个写原生 SQL 的 Flask 应用的例子。

首先实现一个叫作 `cache` 的装饰器，它可以方便地在方法上定义缓存键和缓存时间 (`mc_decorator.py`)。 `cache` 需要一个格式化的函数，它可以把各种需要缓存的参数格式化成成为一个缓存键：

```
import re

__formatters = {}
percent_pattern = re.compile(r'%\w')
brace_pattern = re.compile(r'\{[\w\d\.\[\]\_]+\}')

def formater(text):
    percent = percent_pattern.findall(text)
    brace = brace_pattern.search(text)
    if percent and brace:
        raise Exception('mixed format is not allowed')

    if percent:
        n = len(percent)
        return lambda *a, **kw: text % tuple(a[:n])
    elif '%' in text:
        return lambda *a, **kw: text % kw
    else:
        return text.format

def format(text, *a, **kw):
    f = __formatters.get(text)
    if f is None:
        f = formater(text)
        __formatters[text] = f
    return f(*a, **kw)
```

`format` 函数的格式化效果如下：

```
In : key = 'web_develop:users:%s'
In : id_ = 1
In : format(key % '{id_}', id_=id_)
Out: 'web_develop:users:1'

In : key = 'web_develop:users:%s:%s'
```

```
In : format(key % ('{id_}', '{type}'), id=id_, type='a')
Out: 'web_develop:users:1:a'
```

inspect 模块提供了一些获取活动对象信息的函数，inspect.getargspec 可以获取函数和方法的参数列表。举两个例子：

```
In : f = lambda x: x
In : inspect.getargspec(f)
Out: ArgSpec(args=['x'], varargs=None, keywords=None, defaults=None)

In : f = lambda x, y=10: x + y
In : inspect.getargspec(f)
Out: ArgSpec(args=['x', 'y'], varargs=None, keywords=None, defaults=(10,))
```

通过 getargspec 就可以根据方法的参数获得对应的缓存 key。举个例子：

```
In : def f(id_, type):
...:     return id_, type
...:

In : arg_names, varargs, varkw, defaults = inspect.getargspec(f)

In : gen_key_factory(key, arg_names, defaults)(1, 2)
Out: ('web_develop:users:1:2', {'id_': 1, 'type': 2})
```

gen_key_factory 的代码如下：

```
import inspect

def gen_key_factory(key_pattern, arg_names, defaults):
    args = dict(zip(arg_names[-len(defaults):], defaults)) if defaults else {}
    if callable(key_pattern):
        names = inspect.getargspec(key_pattern)[0]

    def gen_key(*a, **kw):
        aa = args.copy()
        aa.update(zip(arg_names, a))
        aa.update(kw)
        if callable(key_pattern):
            key = key_pattern(*[aa[n] for n in names])
        else:
            key = format(key_pattern, *[aa[n] for n in arg_names], **aa)
        return key and key.replace(' ', '_'), aa
    return gen_key
```

cache 装饰器通过 `gen_key_factory` 获得缓存键，在没有缓存的时候 `set`，缓存未过期前通过 `get` 使用缓存：

```
import time
from functools import wraps

MC_DEFAULT_EXPIRE_IN = 0 # 永不过期

def cache(key_pattern, mc, expire=MC_DEFAULT_EXPIRE_IN, max_retry=0):
    def deco(f):
        arg_names, varargs, varkw, defaults = inspect.getargspec(f)
        if varargs or varkw:
            raise Exception("do not support varargs")
        gen_key = gen_key_factory(key_pattern, arg_names, defaults)

        @wraps(f)
        def _(*a, **kw):
            key, args = gen_key(*a, **kw)
            if not key:
                return f(*a, **kw)
            force = kw.pop('force', False)
            r = mc.get(key) if not force else None

            retry = max_retry
            while r is None and retry > 0:
                # when node is down, add() will failed
                if mc.add(key + '#mutex', 1, int(max_retry * 0.1)):
                    break
                time.sleep(0.1)
                r = mc.get(key)
                retry -= 1

            if r is None:
                r = f(*a, **kw)
                if r is not None:
                    mc.set(key, r, expire)
                if max_retry > 0:
                    mc.delete(key + '#mutex')
            return r
        _._original_function = f
        return _

    return deco
```

```
def create_decorators(mc):
    def _cache(key_pattern, expire=0, mc=mc, max_retry=0):
        return cache(key_pattern, mc, expire=expire, max_retry=max_retry)
    return {'cache': _cache}
```

在 mc.py 文件中，添加如下两行：

```
from decorators import create_decorators
globals().update(create_decorators(mc))
```

相当于把 cache 放进了 mc 的命名空间中。

基于第4章4.2节的 Flask-RESTful 例子，将其修改成使用 Libmc 的版本（app_with_mc.py）。

User 类的代码如下：

```
from sqlalchemy import create_engine
from mc import mc, cache

app = Flask(__name__)
DATABASE_URI = 'mysql://web:web@localhost:3306/r'
api = Api(app)

con = create_engine(DATABASE_URI).connect()

USER_KEY = 'web_develop:users:%s'

class User(object):
    def __init__(self, id, name, address):
        self.id = id
        self.name = name
        self.address = address

    @classmethod
    def add(cls, name, address):
        sql = ('insert into test_user(name, address) '
              'values(%s, %s)')
        id_ = con.execute(sql, (name, address)).lastrowid
        cls.clear_mc(id_)
        return cls.get(id_)

    @classmethod
    @cache(USER_KEY % '{id_}') # 可以添加第二个参数，设置缓存时间
    def get(cls, id_):
        if not id_:
            return None
        row = con.execute(
```

```

        'select id, name, address '
        'from test_user where id=%s', id_).fetchone()
    return cls(*row) if row else None

    @classmethod
    def get_user_by_name(cls, name):
        sql = ('select id from test_user '
              'where name=%s')
        rows = con.execute(sql, name).fetchall()
        return cls.get(*rows[0]) if rows else None

    def delete(self):
        con.execute(
            'delete from test_user where id=%s', self.id)
        self.clear_mc(self.id)

    @classmethod
    def clear_mc(cls, id_):
        mc.delete(USER_KEY % id_)

```

User 将创建、删除和查询封装起来，还添加了清除缓存、通过 name 字段获取 User 实例这两个方法。需要注意类方法和实例方法的使用：delete 只能在 User 实例后才能调用。

UserResource 相对地也有所修改：

```

class UserResource(Resource):
    @marshal_with(resource_fields)
    def get(self, name):
        user = User.get_user_by_name(name=name)
        return user

    def put(self, name):
        address = request.form.get('address', '')
        User.add(name, address)
        return {'ok': 0}, 201

    def delete(self, name):
        user = User.get_user_by_name(name)
        if user:
            user.delete()
        return {'ok': 0}

```

这样就把 Memcached 引用进来了。

缓存更新策略

有两种常见的更新方案：

- 懒惰式加载。客户端先查询 Memcached，如果命中，则返回结果；如果没命中（没有数据或已经过期），则从数据库中获得最新数据，并写回到 Memcached 中，最后返回结果。这种方法直接、简单。但是在高并发的场景下，突然失效会让后端数据库的压力骤增。
- 主动更新。默认缓存永不失效。当有数据需要更新时，同时也会把最新数据写回到 Memcached 中。这种更新如果耗时过长，应该使用异步的更新，如放在消息队列中。

Memcached 的永不失效其实是设置超时时间为 0，当内存不足时，会触发 LRU 机制，删除最近最少使用的内存空间。

Memcached 使用的经验

1. 批量获取时尽可能使用“mc.get_multi”，与同数量的“mc.get”相比，能减少网络请求的次数。
2. 对缓存全部更新，可以直接升级缓存键，在缓存键字符串最后加个版本号，比如上例中的 USER_KEY：

```
USER_KEY = 'web_develop:users:%s:v2'
```
3. 批量更新缓存的时候应该尽量少给后端数据库带来压力，需要对缓存预热。
4. Memcached 不仅能缓存 SQL 查询的结果，它甚至还可以缓存 HTML 片段。在实际工作中笔者曾经用 Memcached 来缓存 Mako Cache 的页面数据，性能更好。
5. 按照 Memcached 协议（<http://bit.ly/28UOu5M>），缓存的值必须小于 1 MB，内容可以是任意字符串。虽然 Libmc 通过拆分支持大于 1 MB、小于 10 MB 的结果，但是不鼓励这么做。
6. 不鼓励使用读/写/删之外的接口。不鼓励使用的接口包括但不限于：append、prepend、incr、decr、add、replace。Memcached 并不适合做上面这些接口做的事，使用这些接口也会给运维带来困难。

键值对数据库 Redis

Redis 是一个基于内存的键值对存储系统，常用作数据库、缓存和消息代理。它支持字符串、字典、列表、集合、有序集合、位图（Bitmaps）、地理位置、HyperLogLog 等多种数据结构，

所以常常被称为数据结构服务器。Redis 支持事务、分片、主从复制，支持 RDB（将内存中的数据保存在文件中）和 AOF（类似于 MySQL 的 binlog）两种持久化方式，还支持订阅分发、Lua 脚本、集群（Redis 3.0 加入的功能）等特性。在用作缓存时 Redis 和 Memcached 功能类似，但它还能做到 Memcached 不能做到的几点：

1. Web 应用中常需要将一些重要数据持久化到硬盘，避免宕机等原因导致数据丢失。Redis 会周期性把更新的数据写入磁盘或者追加到命令日志中，并且在此基础上实现了主从同步。而 Memcached 在进程关闭之后数据就会丢失。
2. 一些业务为了简化工作，需要使用列表、集合这样只有 Redis 才支持的数据结构。相对于 Memcached，Redis 有更多的应用场景。
3. Redis 提供了丰富的命令。比如，可以通过通配符查看线上已经存在的键、判断一个键是否存在（Memcached 这点很不方便，没有设置缓存和设置的缓存为 None 不好区分），方便地获取服务器信息和统计数值（通过 INFO）等。在 Python 客户端中这些功能可以直接集成到项目中，能帮助运维收集服务状态监控数据，绘制性能图表等。

我们先安装 Redis：

```
> sudo apt-get install redis-server -yq
```

安装完毕 Redis 已经启动了，可以验证一下：

```
> redis-cli
127.0.0.1:6379> quit
```

安装 Redis 的 Python 客户端：

```
> pip install redis
```

操作 Redis

看看 Redis 操作列表的用法：

```
In : import redis
In : conn = redis.Redis()
In : conn.rpush('a', '1') # a就是要操作的键
In : conn.lrange('a', 0, -1) # 表示从索引为0的元素到最后一个元素
Out: ['1']
In : conn.rpush('a', '2')
Out: 2L
In : conn.lrange('a', 0, -1)
Out: ['1', '2']
```


可以看到 Redis 直接进行原子操作。Redis 还支持对于列表的其他类型的操作，我们接着演示。

```
In : conn.lpush('a', '3') # 将值推入到列表的左端
Out : 3L
In : conn.lindex('a', 0) # 返回列表中第0个元素的值
Out : '3'
In : conn.rpush('a', *[4, 5, 6]) # 为了测试效果，一次性推入3个元素
Out : 6L
In : conn.lrange('a', 0, -1)
Out : ['3', '1', '2', '4', '5', '6']
In : conn.ltrim('a', 1, 4) # 对列表进行修剪，只保留索引从1到4的值
Out : True
In : conn.lrange('a', 0, -1)
Out : ['1', '2', '4', '5']
In : conn.lpop('a') # 移除并返回列表最左端的元素
Out : '1'
In : conn.rpop('a') # 移除并返回列表最右端的元素
Out : '5'
```

除了列表，另外一个常用的数据结构为字典，在 Redis 中这样使用：

```
In : conn.hset('d', 'a', 1) # 给名字为d的键添加一个名字为a，值为1的字段
Out : 1L
In : conn.hmset('d', {'b': 2, 'c': 3}) # 一次性的添加多个字段
Out : True
In : conn.hget('d', 'b') # 获取字段b的值
Out : '2'
In : conn.hmget('d', ['a', 'b']) # 一次性的获取多个字段的值
Out : ['1', '2']
In : conn.hgetall('d') # 获取d的全部字段和值的对应关系（字典）
Out : {'a': '1', 'b': '2', 'c': '3'}
```

这只是众多数据结构中的最常用的 2 种数据结构提供的主要操作功能，你可以从命令页（<http://bit.ly/28VOVO2>）找到全部命令列表，本书就不一一列举了。但是可以想象，Redis 提供了非常多便利的方法来完成工作。

Redis 应用场景

有哪些场景可以使用 Redis 代替数据库呢？其实原则很简单：当不需要数据库的高级功能（比如事务提供的回滚、关联查询、UPDATE 操作等），且 Redis 能满足此应用场景时就可以选择 Redis。除了缓存，我们举例说明常用的几种场景。

取最新 N 个数据的操作

现在看一下使用 SQLAlchemy、Flask 和 Redis 的例子（latest_files.py）。首先定义一个简单的 PasteFile 模型：

```
class PasteFile(db.Model):
    __tablename__ = 'files'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(5000), nullable=False)
    uploadtime = db.Column(db.DateTime, nullable=False)

    def __init__(self, name='', uploadtime=None):
        self.uploadtime = datetime.now() if uploadtime is None else uploadtime
        self.name = name
```

接着定义两个视图，第一个是更新的视图：

```
r = redis.StrictRedis(host='localhost', port=6379, db=0)
MAX_FILE_COUNT = 50
```

```
@app.route('/upload', methods=['POST'])
def upload():
    name = request.form.get('name')

    pastefile = PasteFile(name)
    db.session.add(pastefile)
    db.session.commit()
    r.lpush('latest.files', pastefile.id)
    r.ltrim('latest.files', 0, MAX_FILE_COUNT - 1)

    return jsonify({'r': 0})
```

r.lpush 表示对列表左侧放入新的数据库条目的 id，r.ltrim 用来修剪列表，只保留最近的 50 个结果。

第二个是获取最近上传文件列表的视图：

```
@app.route('/latest_files')
def get_latest_files():
    start = request.args.get('start', default=0, type=int)
    limit = request.args.get('limit', default=20, type=int)
    ids = r.lrange('latest.files', start, start + limit - 1)
    files = PasteFile.query.filter(PasteFile.id.in_(ids)).all()
    return json.dumps([{'id': f.id, 'filename': f.name} for f in files])
```

先获得最近上传的文件 id 列表，再获得这些模型对象。

验证效果之前先创建一些数据：

```
In : from chapter6.section4.latest_files import app, PasteFile, r
In : import time
In : import random
In : import string
In : with app.test_client() as client:
...:     for _ in range(100):
...:         client.post('/upload', data={'name': ''.join(random.sample(string.
...:             ascii_letters, 10))})
...:         time.sleep(0.5)
...:
```

使用 `r.lrange` 获取最近的 5 个条目：

```
In : start = 0
In : limit = 5
In : ids = r.lrange('latest.files', start, start + limit - 1)
In : ids
Out: ['100', '99', '98', '97', '96']
```

获取条目就是在内存中完成的。使用 SQLAlchemy，就需要用如下方式：

```
In : from sqlalchemy import desc
In : [id for id, in PasteFile.query.with_entities(PasteFile.id).order_by(desc(PasteFile
...:     .id)).offset(start).limit(limit).all()]
Out: [100L, 99L, 98L, 97L, 96L]
```

虽然 `id` 的类型不同，但效果是一样的：

```
In : PasteFile.query.get('1').name
Out: u'gtRQMBePAm'
In : PasteFile.query.get(1).name
Out: u'gtRQMBePAm'
```

这个时候可能你会有疑问，之前使用 Libmc 是直接缓存模型对象的。这里比较曲折，通过 `id` 列表再去获得对应的模型列表。这是因为 Redis 并没有内置序列化工作，如果直接缓存对象，缓存的对象会变成字符串：

```
In : p = PasteFile.query.get(100)
In : p
Out: <latest_files.PasteFile at 0x7f8ad0017350>
In : r.set('a', p)
Out: True
In : a = r.get('a')
In : a
Out: '<latest_files.PasteFile object at 0x7f8ad0017350>'
```

```
In : type(a)
Out: str
```

这涉及序列化/反序列化。将对象的状态信息转换为可以存储或传输的形式过程就是序列化；把这个存储的内容还原成对象就是反序列化。

我们使用 MessagePack 来做序列化和反序列化的工作。MessagePack 是一个基于二进制的高效的对象序列化类库，可用于跨语言通信。它可以像 JSON 那样，在许多种语言之间交换结构对象，但是它比 JSON 更快速也更轻巧。不选择 cPickle 就是希望序列化的数据可以被跨语言使用。

我们先安装 MessagePack：

```
> pip install msgpack-python
```

PasteFile 基于 SQLAlchemy，而且属性 uploadtime 是 Datetime 类型，它们都不支持 MessagePack，需要自定义序列化和反序列化的方法，也就是给 PasteFile 添加两个方法 (lastest_files_with_msgpack.py)：

```
import ast

import msgpack

class PasteFile(db.Model):
    ...

    def to_dict(self):
        # 不一定整个对象的全部方法都要缓存，缓存对象中有用的属性就可以了
        d = {k: v for k, v in vars(self).items() if not k.startswith('_')}
        # datetime格式不能被序列化，需要先转换成对应的时间字符串
        d['uploadtime'] = d['uploadtime'].strftime('%Y%m%dT%H:%M:%S.%f')
        return str(d) # 序列化的数据必须是字符串类型

    @classmethod
    def from_dict(cls, data):
        data = ast.literal_eval(data)
        # id是自增长的字段，没有在__init__中传入，需要在生成对象之后再赋值id这个属性
        id = data.pop('id')
        data['uploadtime'] = datetime.strptime(
            data['uploadtime'], '%Y%m%dT%H:%M:%S.%f')
        p = cls(**data)
        p.id = id
        return p
```

其中 `to_dict` 用来序列化, `from_dict` 用来反序列化。封装之后就是如下函数:

```
def default(obj):
    if isinstance(obj, PasteFile):
        return msgpack.ExtType(42, obj.to_dict())
    raise TypeError('Unknown type: %r' % (obj,))

def ext_hook(code, data):
    if code == 42:
        p = PasteFile.from_dict(data)
        return p
    return msgpack.ExtType(code, data)
```

接下来试验一下:

```
In : p = PasteFile.query.get(100)
In : p
Out: <chapter6.section4.lastest_files_with_msgpack.PasteFile at 0x7eff3030d790>
In : p.name
Out: u'aIMrwLqdAi'
In : packed = msgpack.packb(p, default=default)
In : packed
Out: "\xc7M*{'uploadtime': '20160602T12:34:01.000000', 'name': u'aIMrwLqdAi', 'id': 100
L}"
In : unpacked = msgpack.unpackb(packed, ext_hook=ext_hook)
In : unpacked
Out: <chapter6.section4.lastest_files_with_msgpack.PasteFile at 0x7eff3030d3d0>
In : unpacked.name
Out: u'aIMrwLqdAi'
```

可以看到序列化和反序列化之后, 重要的属性都可以找到。



是否序列化以及如何序列化, 根据业务会有不同的考虑。序列化的优点是可以把对象存放起来, 在对应的位置再反序列化回来, 这也是消息队列能实现的原因。如果想序列化缓存对象, 关键要看对缓存的对象反序列化获得模型和只缓存 id 再获得对应模型的效率孰高孰低。

取 TOP N 操作 (排行榜应用)

豆瓣东西曾经做过一些微信社会化营销的小游戏, 在小游戏结束时立刻列出来分数和排名, 还可以分享给朋友。如果使用传统的 MySQL+Memcached, 思路如下。

1. 创建一张游戏表，表结构如下：

```
CREATE TABLE `game_score`(
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `game_id` smallint(6) unsigned NOT NULL,
  `user_id` int(11) DEFAULT 0,
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `score` float NOT NULL,
  PRIMARY KEY(`id`),
  KEY `idx_score`(`game_id`, `score`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

2. 每当用户完成一次小游戏时，就把对应的游戏 id、用户和得分插入数据库：

```
insert into game_score(game_id, user_id, score) values (GAME_ID, USER_ID, SCORE)
```

3. 通过如下两个 SQL 语句计算得分的排名情况：

```
select count(1) from game_score where game_id = GAME_ID # 获取完成游戏总次数
select count(1) from game_score where game_id = %s and score < SCORE
```

4. 由于插入和查询操作太频繁，对 COUNT 语句添加缓存，插入语句的执行放到异步队列，插入完成后会更新对应的缓存。

这看起来有点复杂，而且插入之后看到的排名由于缓存可能会稍微的延迟。如果使用 Redis 实现，将会比较简单（top_n.py）：

```
import string
import random

import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
GAME_BOARD_KEY = 'game.board'

# 插入1000条随机用户名和分数组成的记录。zadd方法表示我们操作的是一个有序列表
for _ in range(1000):
    score = round((random.random() * 100), 2)
    user_id = ''.join(random.sample(string.ascii_letters, 6))
    r.zadd(GAME_BOARD_KEY, score, user_id)

# 随机获得一个用户和他的得分，zrevrange表示从高到低对列表排序
user_id, score = r.zrevrange(GAME_BOARD_KEY, 0, -1,
                             withscores=True)[random.randint(0, 200)]

print user_id, score

# 获取全部记录条目数
board_count = r.zcount(GAME_BOARD_KEY, 0, 100)
```

```
# 这个用户分数超过了多少用户
current_count = r.zcount(GAME_BOARD_KEY, 0, score)

print current_count, board_count

print 'TOP 10'
print '-' * 20

# 获取排行榜前10位的用户名和得分
for user_id, score in r.zrevrangebyscore(GAME_BOARD_KEY, 100, 0, start=0,
                                         num=10, withscores=True):
    print user_id, score
```

一个有序集合的元素数量可以达到 $2^{32} - 1$ 。使用 `zadd` 的复杂度是 $O(\log(N))$ ，`zrevrange` 和 `zrevrangebyscore` 的复杂度是 $O(\log(N)+M)$ （ N 是 Set 大小， M 是结果/操作元素的个数）。可见使用 Redis 多么方便，效率还很高。

计数器

Redis 非常适合用来做计数器：

```
In : COUNT_KEY = 'id:{id}'.format(id=100)
In : COUNT_KEY
Out: 'id:100'
In : r.get(COUNT_KEY)
In : r.incr(COUNT_KEY)
Out: 1
In : r.incr(COUNT_KEY)
Out: 2
In : r.decr(COUNT_KEY)
Out: 1
In : count = r.incr(COUNT_KEY)
In : count
Out: 2
In : r.incrby(COUNT_KEY, 2)
Out: 4
In : r.incrby(COUNT_KEY, 5)
Out: 9
```

实时统计

Redis 的位图提供了二进制的操作，非常适合存储布尔类型的值。常见场景是记录用户登录状态，用于日后计算一段时间内的活跃用户量。

位图可以对值进行基于二进制的置位操作。我们可以把值的每一位当作一个用户，登录了就置为 1，否则还是默认的 0。假设现有 10 个用户，如果位图的值为 0100100001，表示第 2、5 和 10 个用户是活跃的。用这样的方式来存储非常省内存，而且计算起来很方便，否则需要记录专门的登录日志或者使用表来记录用户的登录情况，这也给数据库服务增加了压力。

我们看一下计算活跃用户数的例子（user_active.py）。首先创建一些登录数据：

```
import redis

ACCOUNT_ACTIVE_KEY = 'account:active'

r.flushall() # 为了测试方便，每次启动后先清理Redis
now = datetime.utcnow()

def record_active(account_id, t=None):
    if t is None:
        t = datetime.utcnow()
    p = r.pipeline() # 使用Redis提供的事务
    key = ACCOUNT_ACTIVE_KEY
    for arg in ('year', 'month', 'day'):
        key = '{}:{}'.format(key, getattr(t, arg))
        p.setbit(key, account_id, 1) # 设置年、月、日三种键
    p.execute() # 事务提交

def gen_records(max_days, population, k): # 随机生成一些数据
    for day in range(1, max_days): # 日期需要从1开始
        time_ = datetime(now.year, now.month, day)
        accounts = random.sample(range(population), k)
        for account_id in accounts:
            record_active(account_id, time_)

gen_records(29, 10000, 2000)
```

最后会添加从 1 号到 28 号的数据，每天都从 10,000 个用户中随机选 2000 个用户登录。

在 IPython 交互模式下试验：

```
> ipython -i user_active.py
# 这个月总的活跃用户数
In : r.bitcount('{}:{}'.format(ACCOUNT_ACTIVE_KEY, now.year, now.month))
Out: 9991 # 总用户为10000，但是有9个用户没有登录过
# 今天的活跃用户数，因为当时设置的是每天2000，这里的值也就是2000
In : r.bitcount('{}:{}'.format(ACCOUNT_ACTIVE_KEY, now.year, now.month, now.day))
Out: 2000
```



```

In : account_id = 1200 # 随机找一个用户
# 查看这个随机用户是否曾经登录过
In : r.getbit('{}:{}'.format(ACCOUNT_ACTIVE_KEY, now.year, now.month), account_id)
Out: 1 # 这个用户在活跃用户中
In : r.getbit('{}:{}'.format(ACCOUNT_ACTIVE_KEY, now.year, now.month), 10001)
Out: 0 # 这个用户肯定不在活跃用户中
# 获取当月1号和2号的键
In : keys = [ '{}:{}'.format(ACCOUNT_ACTIVE_KEY, now.year, now.month, day)
...: for day in range(1, 3)]
# 获取在1号或者在2号的活跃用户数
In : r.bitop('or', 'destkey:or', *keys)
Out: 1250L
# 查询在1号或者在2号的活跃用户数
In : r.bitcount('destkey:or')
Out: 3584
# 获取1号和2号的都活跃的用户数
In : r.bitop('and', 'destkey:and', *keys)
Out: 1250L
# 查询在1号和在2号都活跃用户数
In : r.bitcount('destkey:and')
Out: 416

```

如果位的位置超过了当前字符串的长度，会自动扩充这个字符串。看一下内存占用情况，先编写计算内存使用的函数：

```

def calc_memory():
    r.flushall()

    print 'USED_MEMORY: {}'.format(r.info()['used_memory_human']) # 执行前先看当前内存的占用情况

    start = time.time()

    # 20 * 100000 次 (100万中选择10万)
    gen_records(21, 1000000, 100000)

    print 'COST: {}'.format(time.time() - start) # 记录花费时间
    print 'USED_MEMORY: {}'.format(r.info()['used_memory_human']) # 记录添加记录之后的内存占用情况

In : calc_memory()
USED_MEMORY: 495.86K
COST: 377.873290062
USED_MEMORY: 4.54M

```

通过 calc_memory 的执行结果可以看到使用位图存储非常省空间，200 万的用户活跃计数只

占用了 4 MB 多一点的空间，而且需要强调的是，每次计数我们都是设置了 3 个键。

多了解自己的产品需求和技术难度，当对 Redis 数据结构和数据操作都非常了解熟悉之后，就可以想到非常多的 Redis 应用场景了。

分片和集群管理

之前我们讨论的都是单机 Redis，而在大型网站应用中，热点数据量往往非常大，一个实例可能会放不下。无论是物理主机还是云主机，内存资源都是有限的，这个时候就要考虑横向扩展：由多台主机协同提供服务。现在多核 CPU、几十 GB 内存、SSD 都非常普遍，硬件资源成本却越来越低。当单机的 Redis 实例不能满足需要时，我们可以通过一致性哈希（Consistent Hashing）算法将 Redis 数据的键进行散列，通过哈希函数，让特定的键映射到特定的 Redis 节点上：这就是分片。在 Redis 3.0 之前，需要借助客户端或者代理实现分片。遗憾的是，Redis 的 Python 客户端 redis-py 目前还没有实现这个分片功能。

常见的分片和集群管理方式有如下三种。

1. Twemproxy：它是在 Redis 3.0 之前通用的方式，它是 Twitter 开发的一个支持 Memcached ASCII 和 Redis 协议的、单线程，使用 C 编写的代理。一般一个 Redis 应用会由多个 Twemproxy 来管理，少数 Twemproxy 负责写，多数负责读。通常使用 Redis 自带的 Sentinel 来实现故障的自动切换以达到高可用。Twemproxy 可以定时向 Redis Sentinel 拉取信息，从而替换出现异常的节点。它最大的缺点是无法平滑地扩容/缩容，不便于运维；其次是没有友好的控制面板。需要注意的是，看上去 Twitter 已经不再继续维护它了（<http://bit.ly/28W8aHV>）。
2. Redis Cluster：它是 Redis 3.0 添加的集群方式，也是未来自动分片和高可用的首选方式。这种方式使用数据分片而非一致性哈希来实现，简单地说，就是一个 Redis 集群包含 16,384 个哈希槽，数据库中的每个键都属于这 16,384 个哈希槽的其中一个（使用 CRC16 函数对键获得校验值，对 16384 取余来计算键属于哪个槽）。集群中的每个节点负责处理一部分哈希槽，当添加新的节点时，只需要将对应的某些槽移动到新添加的节点上；当移出节点时，就把这个被移出节点上面的槽分配到其他节点上，而且这个添加和下线的过程不会阻塞整个集群。这个思路很好，只是目前还没有看到大型网站成功的案例，需要继续关注。
3. Codis：豌豆荚在生产环境使用的 Redis 分布式集群解决方案。对于上层的应用来说，连接到 Codis 代理和连接原生的 Redis Server 没有明显的区别（有少量还不支持的命令（<http://bit.ly/28Y4TX3>）），上层应用可以像使用单机的 Redis 一样使用，Codis 底层会处理请求的转发，不停机的数据迁移等工作，所有后边的一切事情，对于前面的客户端来说是透明的，可以简单地认为后边连接的是一个内存无限大的 Redis 服

务。它还兼容 Twemproxy，而且能很容易地把数据从 Twemproxy 迁到 Codis 上，让运维和监控更方便。

NoSQL 数据库 MongoDB

为什么使用 NoSQL

NoSQL 最常见的解释是 Not Only SQL，泛指非关系型的数据库。之前介绍的 Redis 就是一种 NoSQL 数据库。NoSQL 不使用 SQL 作为查询语言，其数据存储不需要预先定义模式，只要保证程序的兼容即可：这在构建 Web 应用程序中尤为有用。它的出现还解决了关系型数据库本身无法克服的一些缺陷：

- 对数据库可以进行高并发读写的需求。
- 对海量数据的高效率存储和访问的需求。
- 高可扩展性和高可用性。

而 NoSQL 相对于传统的关系型数据库的一些不足显得不再那么重要，或者可以通过其他方式来保证：

- 数据库事务一致性需求。
- 数据库的实时读写需求。
- 对复杂的 SQL 查询，特别是多表关联查询的需求。这里需要强调一点，大数据量的 Web 应用都应该减少甚至忌讳使用多个大表的关联查询。

NoSQL 是不能完全替代关系型数据库的，它们没有绝对的孰优孰劣，我们需要根据存储的数据以及操作数据的方式来选用不同的方案。

MongoDB

MongoDB 是一个为当代 Web 应用而生的 NoSQL 数据库，它有如下优点：

1. 文档型存储。可以把关系型数据库的表理解为一个电子表格，列表示字段，每行的记录其实是按照列的字段顺序排列的值的元组。而存储在 MongoDB 中的文档被存储为键-值对的形式，值却可以是任意类型且可以嵌套。之前在用关系数据库的时候，我们需要把产品信息打散到不同的表中，要通过关系表或者使用 join 拼接成复杂的 SQL 语句的方式才能获得需要的数据。现在我们可以更多地把产品信息放在一起，也不需要提前预定产品信息的模式。

2. 使用高效的二进制 BSON 作为数据存储。BSON 是一个类 JSON 的格式，选择 BSON 可以提供更快的遍历速度，提供比 JSON 更多的内置数据类型。
3. 自带高可用及分区的解决方案，分别为副本集（Replica Set）和分片（sharding）。
4. 基于文档的富查询语言。MongoDB 支持动态查询，支持非常多的查询方式，并且可以对文档中的属性建立索引。
5. 内置聚合工具。可以通过 MapReduce 等方式进行复杂的统计和并行计算。
6. MongoDB 在 3.0 之后增加了高性能、可伸缩、支持压缩和文档级锁的数据存储引擎 WiredTiger。官方的性能测试（<http://bit.ly/28SnMaa>）显示，使用新的存储引擎后带来 4~7 倍的性能提升。WiredTiger 从 MongoDB 3.2 开始作为默认的存储引擎，另外它和原来的基于内存映射技术的存储引擎（MMAP）兼容。如果用户将当前自己的应用升级到基于 WiredTiger 的应用，无须更改现有的任何部署，也无须停机进行即可完成。

安装 MongoDB

默认源中没有最新的 MongoDB，需要使用如下方法添加源并安装：

```
> sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
> echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" |
  sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
> sudo apt-get update
> sudo apt-get install -y mongodb-org
```

现在需要手动下载管理脚本：

```
> wget https://github.com/mongodb/mongo/raw/master/debian/init.d
> sudo mv init.d /etc/init.d/mongodb
> sudo chmod +x /etc/init.d/mongodb
```

还需要修改两个内核参数：

```
> sudo sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/enabled'
> sudo sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/defrag'
```

然后启动 MongoDB：

```
> sudo /etc/init.d/mongodb start
```



Vagrant 环境下还可以使用 systemd 管理：

```
> wget https://github.com/mongodb/mongo/raw/master/debian/mongod.service
> sudo mv mongod.service /etc/systemd/system
> sudo systemctl start mongod
```

安装 MongoDB 的 Python 驱动：

```
> pip install pymongo
```



项目的名字是 mongo-python-driver (<http://bit.ly/28WeOf0>)，pymongo 只是包的名字。

使用 pymongo 的例子

我们先看一下基本的 CRUD 的例子 (example_pymongo.py)：

```
import random

import pymongo

client = pymongo.MongoClient('mongodb://localhost:27017/')
client.drop_database('test') # 保证之前没有数据，删除名为test的数据库
db = client.test # 使用test这个数据库
coll = db.coll # 使用coll这个集合

# 插入单条记录
rs = coll.insert_one({'a': 1, 'b': 2})
object_id = rs.inserted_id
print rs.inserted_id # 打印插入的对象id

# 插入多条记录
rs = coll.insert_many([{'a': random.randint(1, 10), 'b': 10}
                       for _ in range(10)])

print rs.inserted_ids # 打印插入的对象id列表

# 查询单条（符合的第一条）记录
print coll.find_one({'a': 1, 'b': 2})

# 集合当前全部文档数
print coll.count()
```

```

cursor = coll.find({'a': {'$lte': 1}}) # 查询结果是一个游标
print cursor.count() # 符合查询的文档数

for r in cursor:
    print r, r['b'] # 打印符合查询的文档内容, 以及其中b键的值

# 注意, 这个循环只能进行一次。如果想再获得查询结果, 需要重新find或者使用list(
    cursor)把结果存起来

# 对查询结果排序
print list(coll.find({'a': {'$lte': 1}}).sort([('b', -1)]))
# -1也可以表示为pymongo.DESCENDING

# 对查询结果可以限制返回文档数, 控制跳过的结果数
print coll.find({'b': {'$gt': 1}}).limit(1).skip(1).next() # next相当于find_one

# 找到后更新, 下面的例子第一个参数是过滤条件, 第二个参数是要更新的操作 (设置b为
    3, a自增长1)
# upsert为True表示找不到会创建一条记录, 可以理解为get_or_create
rs = coll.find_one_and_update({'a': 1, 'b': 2},
                               {'$set': {'b': 3}, '$inc': {'a': 1}},
                               upsert=False)
print rs # 返回更新前的文档
# 同样的还有find_one_and_replace和find_one_and_delete
print list(coll.find({'a': 2, 'b': 3})) # 上述文档已经被更新为这个文档
coll.find_one_and_update({'a': 1, 'b': 2},
                          {'$set': {'b': 3}, '$inc': {'a': 1}},
                          upsert=True) # 虽然没有符合{'a': 1, 'b': 2}的记录, 但是会
                                      新建一个
print coll.find({'a': 2, 'b': 3}).count() # 发现现在有两条文档记录了

# 删除单个文档
coll.delete_one({'a': 2, 'b': 3})

# 一次性删除多个文档

rs = coll.delete_many({'a': 2, 'b': 3})
# 如果没有符合的条目也不会提示, 但是可以通过rs.deleted_count获得删除的数量
print rs.deleted_count

```

MongoDB 的一大核心是聚合功能, 主要用于返回计算好的结果。

常用的执行聚合的方式有如下三种。

1. 使用管道。管道在 UNIX 和 Linux 中一般用于将当前命令的输出结果作为下一个命令的参数：

```
In : cursor = coll.aggregate([
    {'$match': { 'b' : { '$gt': 1, '$lt': 11}}},
    {'$group': {'_id': None, 'count': {'$sum': 1}, 'averageA': {'$avg': '$a'
    }}},
    {'$project': {'_id': 0, 'count': 1, 'averageA': 1}}])
In : cursor.next()
Out: {u'averageA': 5.0, u'count': 10}
```

先通过 “\$match” 获得符合条件的文档，把它传给第二个 “\$group” 来分组，分组会计算总数（count）和 a 的平均值（averageA），然后传给第三个 “\$project” 过滤掉字段 _id。这里需要注意使用管道时的一些利弊，可以参考 Aggregation Pipeline Optimization (<http://bit.ly/28WbHGk>)。

2. MapReduce。Map-Reduce 是一种计算模型。它将大批量的工作（数据）分解执行，然后再将结果合并成最终结果（Reduce）。这种模式下使用原生的 JavaScript 函数：

```
In : from bson.code import Code
In : mapper = Code("""function () {
...: var key;
...: if (this.a < 3) {
...: key = 'lt 3'; // 根据a的值设置不同的键
...: } else {
...: key = 'gte 3';
...: }
...: emit(key, 1);}""")
In : reducer = Code("""function (key, values) {
...: return Array.sum(values)}""")
In : rs = coll.map_reduce(mapper, reducer, 'c', query={ 'b' : { '$gt': 1, '$lt':
    11}}) # c是存放计算结果的集合的名字。query用来过滤需要计算的文档，符
    合条件的才会做MapReduce计算
In : for i in rs.find():
...:     print i
...:
{u'_id': u'gte 3', u'value': 8.0} # _id就是上面的键，符合大于或等于3的文档
    有8个
{u'_id': u'lt 3', u'value': 2.0} # 符合小于3的文档有2个
```

如果这样的 JavaScript 函数或者变量非常常用，可以存放在服务器上以备重复使用：

```
In : db.system.js.insert_one({'_id': 'mapper', 'value': mapper})
In : rs = coll.map_reduce(db.eval('mapper'), reducer, 'c')
```

3. 单一目的的聚合操作方法。MongoDB 目前已经提供了 count、distinct 和 group 等聚合方法，可以直接调用：

```
In : coll.distinct('a') # 找出a键的所有不同的值
Out: [8, 5, 7, 3, 6, 2, 1]
```

这就是聚合的魅力。聚合运算符的种类非常多，可以在 Aggregation Pipeline Operators (<http://bit.ly/294MTYm>) 找到全部的运算符。

使用 Mongoengine 的例子

在前面我们曾经使用 MySQL 的 ORM 框架 SQLAlchemy，同样的 MongoDB 也有自己的 ODM（对象文档映射）框架，最知名的是 Mongoengine。如果了解甚至使用过 SQLAlchemy 的话会发现它会非常容易理解和上手。

本节把文件托管服务的模型改成使用 Mongoengine。并不是说我们使用 Flask 和 Mongoengine，就要安装 Flask-Mongoengine，虽然 Flask-Mongoengine 集成了 WTForms，也提供了一些其他功能，但是对我们这个应用没有太大意义。我们将直接使用 Mongoengine 来实现，这样也可以了解扩展隐藏起来的一些细节。

先安装它：

```
> pip install mongoengine
```

现在把第 3 章 3.6 节的文件托管服务改成使用 Mongoengine。改动分为如下三部分：

- 替换 PasteFile 模型。
- 修改模型的方法。
- 修改视图中保存 PasteFile 模型实例的方法。

首先替换定义的模型：

```
from mongoengine import (
    Document as BaseDocument, connect, ValidationError, DoesNotExist,
    QuerySet, MultipleObjectsReturned, IntField, DateTimeField, StringField,
    SequenceField)
# 连接本机27017端口的MongoDB服务器的数据库r
connect('r', host='localhost', port=27017)

class BaseQuerySet(QuerySet):
    # 当捕捉到多个值/不存在或者验证失败这三种异常时，直接返回404，这个方法很常用
    def get_or_404(self, *args, **kwargs):
        try:
            return self.get(*args, **kwargs)
```



```

        except (MultipleObjectsReturned, DoesNotExist, ValidationError):
            abort(404)

class Document(BaseDocument):
    meta = {'abstract': True,
           'queryset_class': BaseQuerySet} # 使用自定义的BaseQuerySet

class PasteFile(Document):
    id = SequenceField(primary_key=True)
    filename = StringField(max_length=5000, null=False)
    filehash = StringField(max_length=128, null=False, unique=True)
    filemd5 = StringField(max_length=128, null=False, unique=True)
    uploadtime = DateTimeField(null=False)
    mimetype = StringField(max_length=128, null=False)
    size = IntField(null=False)
    meta = {'collection': 'paste_file'} # 自定义集合的名字

```

需要说明的是，MongoDB 默认使用是 `_id` 这个字段作为主键，但是它是 `bson.ObjectId` 的实例，不方便 `short_url` 模块获得短链接，而使用 `SequenceField` 替换它，就可以达到 `id` 自增长的目的了。

`PasteFile` 的 `__init__` 方法中需要初始化父类的 `__init__` 方法：

```

def __init__(self, filename='', mimetype='application/octet-stream',
             size=0, filehash=None, filemd5=None, *args, **kwargs):
    super(PasteFile, self).__init__(filename=filename, mimetype=mimetype,
                                    size=size, filehash=filehash,
                                    filemd5=filemd5, *args, **kwargs)
    self.uploadtime = datetime.now()
    ...

```

`PasteFile` 中的三个 `get_by*` 方法中获得模型实例的方式也要修改一下：

```

@classmethod
def get_by_symlink(cls, symlink, code=404):
    id = short_url.decode_url(symlink)
    return cls.objects.get_or_404(id=id)

@classmethod
def get_by_filehash(cls, filehash, code=404):
    return cls.objects.get_or_404(filehash=filehash)

@classmethod
def get_by_md5(cls, filemd5):
    rs = cls.objects(filemd5=filemd5)

```

```
return rs[0] if rs else None
```

最后修改视图中的保存方法，原来使用：

```
db.session.add(paste_file)
db.session.commit()
```

现在只需要 save 就可以了：

```
paste_file.save()
```

这样就替换完成了。

MongoDB 实践经验

善用组合式的大文档

可以尽量把数据组合起来节省空间，提高读的性能。MongoDB 的一个应用场景是实时分析。假设我们每分钟都要记一次日志，内容格式如下：

```
{ metric: "review_count", client_type: 2, value: 23, date: ISODate("2016-03-22 10:10")
}
{ metric: "review_count", client_type: 2, value: 9, date: ISODate("2016-03-22 10:11") }
```

采用组合的方式，例如把一个小时的日志放到一个文档中：

```
{ metric: "review_count", client_type: 2, date: "2016-03-22", hour: 10, 10: 23, 11: 9)
}
```

可以通过如下的测试对比一下（use_big_documents.py）：

```
import random
from datetime import datetime, timedelta

import pymongo

RANGE = 1440
l = [random.randint(1, 100) for i in range(RANGE)] # 模拟一天的数据
client = pymongo.MongoClient("localhost", 27017)
client.drop_database('test') # 保证之前没有数据
db = client.test

# 传统方案
for i in range(RANGE):
    s = {'metric': 'review_count', 'client_type': 2, 'value': l[i],
        'date': datetime(2016, 03, 22, 0, 0) + timedelta(minutes=i)}
```

```
db.a.insert_one(s)
```

单个大文档（每小时一个文档）的方案

```
for i in range(24):
    s = {'metric': 'review_count', 'client_type': 2, 'date': '2016-03-22', 'hour': i}
    s.update({str(k): v for k, v in enumerate(l[60 * i:60 * (i + 1)])})
    db.b.insert_one(s)
```

```
def get_hour_data1(hour):
    return [i['value'] for i in db.a.find({
        'date': {'$gte': datetime(2016, 3, 22, hour, 0),
                    '$lt': datetime(2016, 3, 22, hour + 1, 0)})}]
```

```
def get_hour_data2(hour):
    c = db.b.find_one({'hour': hour})
    return [c[str(i)] for i in range(60)]
```

对比一下性能：

```
In : db.eval('db.a.stats()["size"]')
Out: 128160.0
```

```
In : db.eval('db.b.stats()["size"]')
Out: 13560.0
```

```
In : %timeit -n 10000 get_hour_data1(1)
10000 loops, best of 3: 1.05 ms per loop
```

```
In : %timeit -n 10000 get_hour_data2(1)
10000 loops, best of 3: 311 µs per loop
```

按月查询所占空间只有原来的 8% 左右，花费的时间不到原来的 1/3。

正确使用索引

添加索引是为了提高查询速度。但是无论代码评审做得多好，缺少索引、索引错误、索引太多这三类问题总还是会有漏网之鱼，笔者的经验是经常关注 MongoDB 的日志，另外每周一都会收到上周慢查询 TOP100 的邮件（MySQL 的慢查询也会有类似的邮件），从其中发现问题并寻找优化的可能，及时修复。举个实际的例子（预先去掉一个正确的索引）：

```
In : import pymongo
In : client = pymongo.MongoClient()
```

```
In : db = client.products
In : c = db.promo_status.find({'symbol': 1}).sort('create_date', -1).limit(1)
In : c.next()
```

这个时候能感觉执行 `next` 的时候很慢。MongoDB 默认会记录用时超过 100 ms 的慢查询（可以通过 `slowms` 参数修改这个值），这时候看 `/var/log/mongodb/mongod.log` 新增了这样一条记录：

```
2016-05-31T00:08:17.267+0800 I COMMAND [conn10] command products.promo_status command:
  find { find: "promo_status", filter: { symbol: 1 }, sort: { create_date: -1 },
  limit: 1 } planSummary: COLLSCAN keysExamined:0 docsExamined:6318177 hasSortStage:1
  cursorExhausted:1 keyUpdates:0 writeConflicts:0 numYields:73611 nreturned:1 reslen
:296 locks:{ Global: { acquireCount: { r: 73612 } }, MMAPV1Journal: { acquireCount:
  { r: 73612 } }, Database: { acquireCount: { r: 73612 } }, Collection: {
  acquireCount: { R: 73612 } } } protocol:op_query 34162ms
```

通过日志确实是可以发现问题的，但是需要处理日志，操作起来不太方便。可以打开分析器，把慢查询记录到 `system.profile` 集合中，方便最后对结果进行处理（可以通过 `profile` 参数指定）。需要注意 `system.profile` 是一个 1 MB 的固定集合，当集合达到这个固定值之后新文档就开始覆盖最早的文档。根据需要，至少要保存一周的慢查询，所以最好设置成 10 MB 集合。如果你的项目问题非常多，那么需要再加大这个值：

```
In : db.set_profiling_level(0)
In : db.system.profile.drop()
In : db.create_collection('system.profile', capped=True, size=10000000)
In : db.set_profiling_level(1)
In : db.profiling_level()
Out: 1
```

现在就可以获得我们想要的分析结果了：

```
# 获得上一周花费时间TOP100的慢查询日志
In : db.system.profile.find({'ts': {'$gte': datetime(2016, 3, 14), '$lt': datetime
  (2016, 3, 21)}}).sort('millis', -1).limit(100)
# 获得数据库products的promo_status集合，花费时间超过300 ms的慢查询日志
In : db.system.profile.find({'ns': 'products.promo_status', 'millis': {'$gt': 300}})
```

还可以聚合：

```
In : from bson.code import Code
In : reducer = Code("""
....: function(obj, prev){
....:     prev.count++;
....: }""")
In : db.system.profile.group(key={'ns': True}, condition={}, initial={'count': 0},
  reduce=reducer) # 聚合不同的集合的慢查询数量
```

```
In : db.system.profile.group(key={'ns': True}, condition={},
initial={'millis': 0}, reduce=Code('function(obj, prev){ prev.millis += obj.millis;}'))
# 聚合不同集合的慢查询花费的总时间
```

现在分析下之前提到的那个慢查询：

```
In : rs = c.explain()
In : rs['queryPlanner']['winningPlan']['inputStage']['inputStage']['stage']
Out: u'COLLSCAN' # 这是一个全集合扫描
In : rs['executionStats']['totalDocsExamined']
Out: 6318177 # 扫描的文档数量，也就是全表的文档数量
In : rs['executionStats']['nReturned']
Out: 1 # 但是只返回了一个结果
```

很显然，这个查询的结果数和需要扫描的文档数的比例完全不合理，需要创建索引。创建索引有以下几条规则：

1. 通常创建的联合索引是按照查询条件的顺序来的。
2. 排序字段通常放在联合索引的最后。但是涉及范围查询时可能要把排序字段提前，需要根据业务实际情况作对比。
3. 单个字段建索引时，不需要考虑索引升序还是降序。但是当数据量很大，建联合索引时，需要注意索引升序和降序的查询效率。
4. 把能过滤数据量多的字段放在前面。

创建一个标准索引：

```
In : db.promo_status.create_index([('symbol', 1), ('create_date', -1)])
Out: u'symbol_1_create_date_-1'
```

如果生产环境的数据量很大，可以在后台创建索引，减少对用户的影响：

```
db.promo_status.create_index([('symbol', 1), ('create_date', -1)], background=True)
```

现在再分析之前的查询：

```
In : rs = c.explain()
In : rs['queryPlanner']['winningPlan']['inputStage']['inputStage']['stage']
Out: u'IXSCAN' # 使用索引的扫描
In : rs['executionStats']['totalKeysExamined']
Out: 1 # 只扫描了一个索引条目就找到了
In : rs['executionStats']['totalDocsExamined']
Out: 1 # 只扫描了一个文档就找到了
In : rs['executionStats']['executionTimeMillis']
Out: 0
```

执行时间因为太短，显示为 0 s，从 34,162 ms 锐减到 0 ms，这都是索引的功劳。

除此之外，我们还要关注索引字段的顺序。MongoDB 和传统数据库一样，是采用 B 树作为索引的数据结构。对于树形的索引来说，保存热数据使用到的索引在存储上越集中，索引浪费的内存也越小。我们对比如下两种创建索引的方法：

```
db.promo_status.create_index([('symbol', 1), ('create_date', -1)])
```

```
db.promo_status.create_index([('create_date', -1), ('symbol', 1)])
```

使用后者可以保证插入速度的稳定，因为新数据更新索引时只是在索引的尾巴处进行修改，那些插入时间过早的索引在后续的插入操作中几乎不需要进行修改，但是对于我们的查询需求来看，它要扫描的索引条目更多：

```
In : rs = db.promo_status.find({'symbol': 1}).sort('create_date', -1).limit(1).hint([('create_date', -1), ('symbol', 1)]).explain() # hint可以强迫使用特定索引来查询
In : rs['executionStats']['totalKeysExamined']
Out: 1211 # 需要扫描1211个条目
In : rs['executionStats']['executionTimeMillis']
Out: 1 # 花费了1 ms，比之前的慢
```

1 ms 还是可以接受的，这个时候需要根据业务做出取舍。

MongoDB 的索引也遵循“最左前缀”原则，上面创建了 “[('symbol', 1), ('create_date', -1)]”这个复合索引，就不需要再创建 “[('symbol', 1)]”这个索引了（如果存在的话，可以删掉）。

高可用方案

复制（repliation）是大多数数据库系统的标配，即在三台服务器上保存数据的副本。因为故障是无法避免的，复制可以保证生产环境的数据在发生故障之后仍然保持可用状态。故障常见于如下场景：

- 服务器硬件故障。
- 计划的停机，比如服务器硬件维护、升级系统、切换新服务器等。有时候停机重启时可能会出现某些问题，比如新安装的软件或者硬件和系统不兼容。
- 异常断电。断电非常有可能损坏数据，而且需要时间来恢复。

在生产环境中使用复制是非常明智的。MongoDB 提供两种复制，主从复制和副本集。两者的复制机制相同，但是后者提供自动的故障转移：主服务器因为某些原因下线后，如果可能的话，可以把一个从节点提升为主节点。副本集是推荐的复制策略，不要使用主从复制。我们先来看副本集节点的常见角色类型。

- **Primary**: 主节点, 处理客户端的请求, 一般是读写请求。
- **Secondary**: 从节点, 一般有多个。它们各自保存主服务器的数据副本, 主服务器出问题时其中的一个从节点可提升为新主节点, 可以处理客户端的读请求。
- **Arbiter**: 仲裁节点, 不保存数据, 只参与选举, 不复制数据。如果副本集有偶数个节点, 可以增加一个投票节点以保证其成员个数为奇数。这样才能可以正常地选举出主节点。仲裁节点所需资源很少, 可以用做其他用途。
- **Hidden**: 隐藏节点, 只用于备份节点, 不处理客户端的读请求。
- **Secondary-Only**: 只能作为从节点, 主要防止这种性能不高的节点成为主节点。

常见的副本集架构如下:

- 一个主节点用于读写, 两个从节点 (或者一个从节点, 一个仲裁节点), 从节点可读。这是最小的配置, 适用于规模比较小的应用。
- 一个主节点用于写, 多个从节点用于读, 一个从节点用于备份。从节点的数量和业务相关, 从节点的数量越多, 可发生的故障转移的次数就越多。目前副本集最多能有 50 个节点, 当业务需要超过 50 个节点时, 才要用到主从复制的架构。在异地分布式架构中, 用于备份的从节点一般在另一个数据中心。读写分离是为了让主数据库处理增、改、删这样的写操作, 而从数据库处理查询这样的读操作。

需要明确的是, 复制不是备份。复制并不能避免人为操作的错误。例如管理员突然删除了产品数据, 或者部署了错误版本的应用程序代码以致搞乱了部分或者全部数据。所以, 必须要有一个能够让我们从这种场景中恢复数据的备份。

分片方案

随着数据量的增加, 应用对读写吞吐量的要求越来越高, 单台服务器总会遇到内存、CPU、磁盘空间等的瓶颈。这时候就可以将数据库用某种规则分开, 存储在多台服务器上, 这就是分片。可以预想, 分片能让集合扩充到无限大。分片没必要考虑得太早, 但是当需求超过单台服务器能提供的 70% (经验值) 的时候, 就应该开始考虑分片, 并准备实施了。

MongoDB 支持自动分片。它的分片是把一个集合根据片键分成多个块 (chunk, 一个 chunk 包括这个片键区间的数据), 然后将各个块分别分发到各个分片副本集上。所以, 片键的选择非常重要。

1. 不要单独使用 `_id` 或者时间戳这种有递增特性的属性作为片键, 它们只会一直往最后一个副本集中添加数据。
2. 使用随机性高且基数比较大的片键, 这样避免一个分片承受绝大多数负载, 并保证

了分片间数据的均衡。

3. 使用多个属性作为片键。如果没有随机性高且基数比较大的片键，可能会造成巨大块，这时候可以和 `_id` 一起作为片键。
4. 使用数据文档 `_id` 的哈希值，可通过 “`db.collection.create_index([['_id', 'hashed']])`”。如果业务上实在找不到符合的片键，这是一个不错的方案。但是它有一个缺点，对多个文档的查询必将命中所有的分片。

大型网站架构经验

限于篇幅，本书无法把网站架构演进的过程都呈现出来，而且一个网站的架构往往和业务有关，并不适合全部的场景。图 6.1 所示是一个比较通用的大型网站的架构。

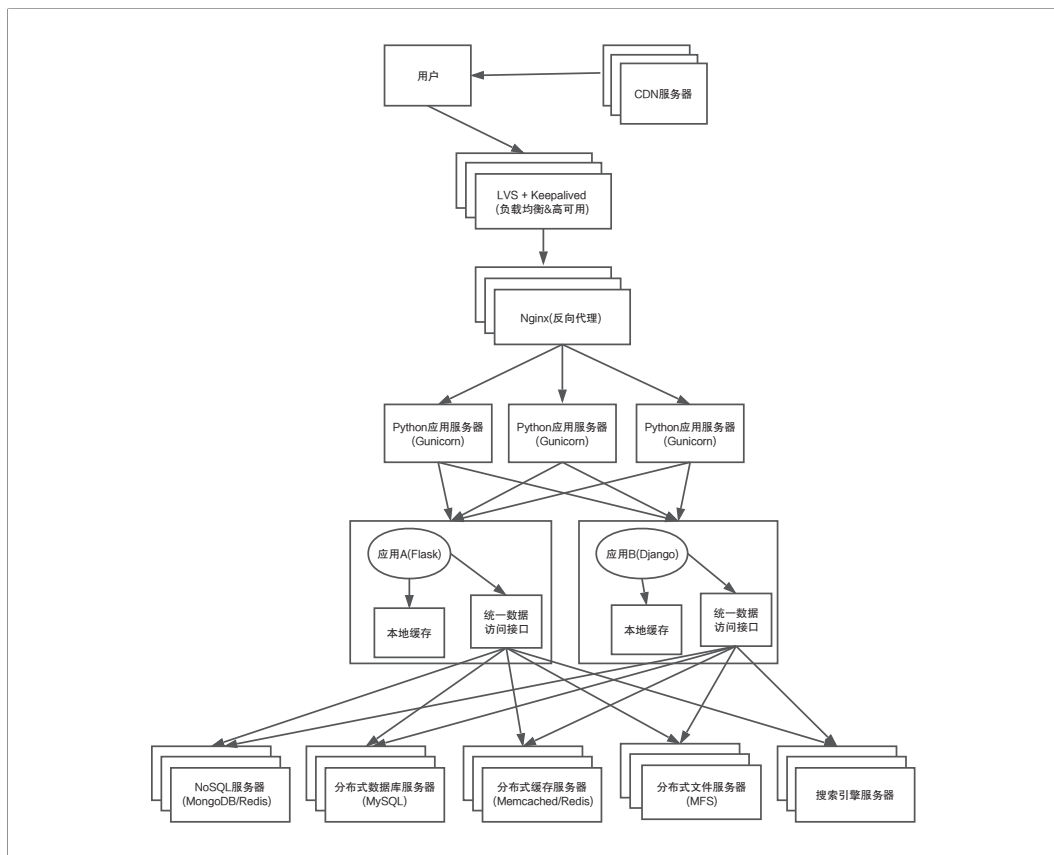


图 6.1 通用的大型网站的架构

图 6.1 中的架构模式主要包含如下部分。

缓存

通常来说，用户访问的热点数据只集中在一小部分数据上，这部分数据应该放在缓存中，这样就减轻了后端应用和数据存储的负担。缓存主要分为如下4种。

1. 本地缓存。在应用服务器内缓存热点数据，当请求访问热点数据时，可以直接使用本机的内存获得结果而不需要访问数据库。
2. 分布式缓存。大型网站的数据量非常庞大，单机无法承受，本地缓存在网站发展到一定程度后还需要分布式缓存集群，应用程序通过网络访问缓存数据。
3. 反向代理。代理服务器位于网站机房之内，用户访问的地址其实是代理服务器地址，由代理服务器决定是从后端换取相应的内容，还是把缓存的内容返回给客户端。通常还会使用 Varnish 来加速应用。
4. CDN（内容发布网络）缓存。发布商出售缓存服务给需要的网站，这些网站不必自己维护缓存工作，只是按需求把内容提交到 CDN 发布商即可。通常 CDN 发布商的缓存服务器分布很广，能自动判断访问者的 IP 地址并解析出对应的 IP 地址，选择最优的缓存服务器响应请求。在 CDN 上缓存网站的一些变化少的内容（如静态资源）可以有效提升页面加载速度。

在实际场景中应该使用不同的缓存设计，提高用户体验。

负载均衡

负载均衡技术是构建大型网站必不可少的架构策略之一。后端服务器的硬件资源可能在 CPU 个数和性能、服务器内存、服务器质量等方面是不一样的，较好的服务器理当负担更大的访问量，较差的服务器理当负担较小的访问量。通过负载均衡可以把用户的请求分发到多台后端的设备上，这样就均衡了服务器的负载，以达到最优化资源使用、最大化吞吐率、最小化响应时间的目的。常见的负载均衡工具包括 LVS、HAProxy、Nginx，对于中小型的 Web 应用它们都可以胜任。但对于复杂的应用场景，在选择负载均衡工具时需要注意如下几点：

- Nginx 仅能支持 HTTP、HTTPS 和 Email 等协议，适用场景最少。HAProxy 和 LVS 都可以对 MySQL 进行负载均衡，但是 Nginx 不可以。
- LVS 不支持正则表达式处理，不能做动静分离。
- HAProxy 配置选项很多，也最灵活，适用场景最多。
- Nginx 对网络的依赖非常小，但是 LVS 对网络的依赖很重。

高可用

之前讨论的都是单个节点的场景，但是单个节点无法满足日益增长的业务压力，这势必需要让服务运行在多个节点上。互联网服务都希望 7*24 不间断，这就要求系统具有高可用性，即组成系统的某些设备宕机或者组件失效时，并不会中断服务。Keepalived 最初是 LVS 的扩展项目，通过对服务器池对象的健康检查，实现对失效机器/服务的故障隔离和负载均衡器之间的切换。类似的工具还有 Heartbeat。但 Keepalived 不会单独出现，而是与其他负载均衡技术一起工作来实现集群的高可用。

业务拆分

大型网站业务场景复杂，为了提高网站可扩展性，需要考虑业务或者产品拆分，分归到不同的业务团队中负责。每个应用独立开发、部署、维护，应用之间通过网络进行通信，或者使用消息队列进行分发。业务拆分还包括对数据库的分库，其实现的成本低，但是效果相对较好。

集群

使用集群是解决高并发、海量数据问题的常用手段。当单台服务器的存储空间和处理能力不能满足要求时，不要盲目提高服务器的配置，而是选择增加一台服务器来分担原服务器的访问和存储压力。

用户看到的大型网站的大部分页面一般都不是单独的应用提供服务。整个页面事实上被分割成了多个部分，每个部分都由独立部署的一个服务器集群来提供服务。集群包含提供相同服务的多个服务器，这既提高了并发能力，也能在某台服务器发生故障时，使用系统失效机制或者按照负载均衡设置将请求转发到其他正常的服务器上，使用户不受影响。

有一点需要注意，不同用途的服务器对硬件资源的要求是不一样的：应用服务器要处理大量的业务逻辑，需要更快的 CPU；文件服务器存储大量用户文件，需要更大的硬盘；数据库服务器需要快速的写入数据和检索到要查询的数据缓存，需要更快的硬盘和更大的内存；缓存服务器需要快速获取数据缓存，只需要更大的内存，等等。应该根据用途来决定采购的标准。

Web 前端性能的优化有如下三点值得考虑。

减少页面请求和请求内容量

HTTP 是无状态的应用层协议，每次请求都需要在客户端/服务端启动独立的线程去处理，开销很昂贵，可以通过合并 CSS、合并 JavaScript、合并图片来实现减少 HTTP 请求的数目。

其中合并图片的方法主要有 CSS Image Sprites（将多张图片合并成一幅单独的图片）、内联图片（通过使用 data:URL 模式而无须任何额外的请求）、IconFont（图标字体）这三种。请求的资源的内容越大，和服务端的连接时间就越长。生产环境要使用被压缩的 JavaScript 和 CSS 资源以提高响应速度。

动静页面分离

把像 JavaScript、CSS 这样的静态资源应用部署在专门的服务器集群中，在页面的模板上通过不同的域名引用。

动态页面静态化

如果动态页面访问量很大但是更新不频繁，可以将其生成一个静态页面，利用静态页面的优化手段加速用户访问，比如浏览器缓存、CDN 等。静态化之后可以有效减轻应用服务器的压力。豆瓣电影年度榜单（<http://bit.ly/2a1IpoE>）就是一个静态页面，数据效果靠 JavaScript 来实现。Mako 虽然也支持模板级别的缓存，但是通过试验对比，还是直接存入 Memcached 的访问效率更高。

第 7 章

系统管理

Python 的一大应用方向就是系统管理，有非常多的自动化运维工具、管理脚本都是使用 Python 编写的。在 Web 开发中或多或少的都会涉及一些系统管理的工作，甚至有很多公司的 Web 开发人员会负责部署、回滚、解决线上环境问题等工作。Web 开发人员非常有必要熟悉系统管理中的主流工具及使用方法。

本章包含如下内容：

- 使用 Supervisor 管理进程。
- 使用 Fabric 进行应用部署。
- 通过部署 Redis 了解配置管理工具 SaltStack 和 Ansible。
- 使用 Psutil 获取系统 CPU、内存、硬盘和网络等信息。
- 配图演示 Sentry 的安装和收集错误信息的效果。
- 使用 StatsD、Graphite、Diamond 和 Grafana 搭建 Web 监控，并介绍常见的运维监控工具及其主要应用场景。

进程管理 Supervisor

Supervisor 是一个用 Python 实现的进程管理工具，可以很方便地启动、重启、关闭、查看进程（不仅仅是 Python 进程）。除了对单个进程的控制，它还可以同时操作多个进程。除此之外它还能监控进程，当进程由于某种原因崩溃或者误操作杀掉后，自动重启并发送事件通知。

我们先安装它：

```
> pip install supervisor
```

Supervisor 组件

Supervisor 包含如下 4 种组件。

1. **Supervisord**：服务端程序，它的主要功能是启动 Supervisord 服务及其管理的子进程，记录日志，重启崩溃的进程等。
2. **Supervisorctl**：命令行客户端程序，它提供一个类似 Shell 的接口，通过 UNIX 域套接字或者 TCP/IP 套接字使用 XML_RPC 协议与 Supervisord 进程进行数据通信。它的主要功能就是管理（启动、关闭、重启、查看状态等）子进程。
3. **Web Server**：实现了在界面上管理进程，还能查看进程日志和清除日志。Web Server 其实是通过 XML_RPC 来实现的，可以向 Supervisord 请求数据。它配置在 [inet_http_server] 块里面。
4. **XML_RPC 接口**：可以通过 XML-RPC 协议对 Web Server 进行远程调用，达到和 Supervisorctl 以及 Web Server 一样的管理功能。

配置 Supervisor

Supervisor 的配置文件一般都叫作 supervisord.conf。启动 Supervisord 的时候会按照如下的路径寻找配置文件：

- 当前目录下的 supervisord.conf (\$CWD/etc/supervisord.conf)。
- 当前目录的 etc 目录下的 supervisord.conf (\$CWD/etc/supervisord.conf)。
- 相对于可执行文件 supervisord 的上一级的 etc 目录下的 supervisord.conf (../etc/supervisord.conf)。
- 相对于可执行文件 supervisord 的上一级的 supervisord.conf (../supervisord.conf)。



可以使用 -c 选项指定不符合如上要求的配置文件路径。

先创建一个统一的存放日志的目录，并让 ubuntu 这个用户成为其拥有者：

```
> sudo mkdir /var/log/supervisord
> sudo chown ubuntu:ubuntu /var/log/supervisord -R
```

下面的例子将通过 Supervisor 管理 Memcached 和 Gunicorn:

```
[unix_http_server]
file=/tmp/supervisor.sock ; 监听HTTP/XML-RPC请求。分号表示之后的内容是注释
;username = dongwm ; 登录管理后台的用户名
;password = 123 ; username和password在安全的网络中不需要设置，所以都是注释的

[inet_http_server] ; 提供Web管理界面
port = 0.0.0.0:5000 ; Web管理后台运行的IP和端口，需要考虑安全性
username = dongwm
password = 123

[supervisord]
logfile=/var/log/supervisord/supervisord.log ; 日志文件
logfile_maxbytes=50MB ; 日志文件大小限制，超过会切分。设置
    为0表示不限制
logfile_backups=20 ; 切分后的日志保留的份数
loglevel=error ; 日志级别，其他可选项为info, debug,
    warn, trace
pidfile=/var/run/supervisord.pid
nodaemon=false ; 使用daemon的方式启动
minfds=1024 ; 可以打开的文件描述符的最小值
minprocs=200 ; 可以打开的进程数的最小值
user=ubuntu ; 启动supervisord进程使用的用户，虽然
    默认就是当前用户，但是指定user是一个好习惯

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///tmp/supervisor.sock ; 使用UNIX域套接字的方式，文件路径必须
    和unix_http_server里面的设定匹配
prompt=web_develop ; 定义提示文本，常用来区分不同的环境。使用清晰的提示能减少操作
    出错

[include] ; 包含其他的配置文件，本例因为演示才把全部配置放在了一起，如果管理的进
    程较多，应该按一定规则分散到不同的配置文件中
files = /etc/supervisor/conf.d/*.conf

[program:app_chapter6] ; 每个program就是一个（组）进程
command=gunicorn -w 3 chapter6.section1.run:app -b 0.0.0.0:9000 ; 启动命令
autostart = true ; 在Supervisord启动的时候也自动启动
autorestart = true ; 程序异常退出后自动重启
startsecs = 5 ; 启动5秒后没有异常退出，就当作已经正常启动了
startretries = 3 ; 启动失败自动重试次数，默认是3次
user=ubuntu
priority=100 ; 优先级设置。低优先级会先启动，后关闭。应用进程应该是优先级最高
```

`redirect_stderr=true` ; 把错误日志重定向到输出的日志中。当然可以把错误日志分开存放, 需要使用`stderr_logfile`参数
`stdout_logfile=/var/log/supervisord/chapter6.log` ; 指定输出的日志的文件路径
`directory=$(ENV_HOME)s/web_develop` ; 启动时会先切换目录进来, 保证启动的时候的相对路径正确性
`stdout_logfile_maxbytes=200MB` ; 输出日志文件大小限制, 超过会切分。设置为0表示 unlimited

`[program:memcached]`
`priority=10`
`numprocs=2` ; 使用进程组
`numprocs_start=2` ; 进程组的数从2开始计数, 因为`numprocs`是2, 也就是使用2和3。如果不指定则是0和1
`process_name=1121%(process_num)s` ; 当`numprocs>1` 进程名字就需要带`process_num`变量
`command=memcached -m 64 -p 1121%(process_num)d -l 127.0.0.1` ; 启动多个进程, 由于`process_num`不同而启动命令不同
`redirect_stderr=true`
`stdout_logfile=/var/log/supervisord/memcached.log`
`stopasgroup=false` ; 如果设置为 `true`, 当进程收到`stop`信号时, 会自动将该信号发给该进程的子进程
`killasgroup=false` ; 如果设置为 `true`, 当进程收到`kill`信号时, 会自动将该信号发给该进程的子进程

`[eventlistener:listener]` ; Supervisor提供事件监听和通知机制
`buffer_size=30` ; 增大事件队列的长度, 如果监控的事件量大, 不能处理的事件会被抛弃
`priority=-1` ; 事件监控的优先级应该最低
`command=python $(ENV_HOME)s/web_develop/chapter7/section1/listener.py`
`events=PROCESS_STATE_EXITED,PROCESS_STATE_STOPPED,PROCESS_STATE_FATAL,TICK_60` ; 检测如上三种事件, 全部事件类型可以查看[Event Types](http://supervisord.org/events.html#event-types)
`stdout_logfile=/var/log/supervisord/listener.log`
`stderr_logfile=/var/log/supervisord/listener.log` ; `eventlistener`类型不能使用
`redirect_stderr=true`
`stdout_events_enabled=false`
`stderr_events_enabled=false`

Supervisor 的注释要使用分号。在上述配置中, `program` 块中指定了要管理的进程, 但是要注意, 被管理的进程不能使用 `daemon` 模式, 必须在前台运行。

`ENV_HOME` 是 Supervisor 内置的变量之一, 表示当前用户的家目录。

在 `eventlistener` 块中使用了自定义的事件监听脚本 `listener.py`, 它用来把事件日志写到文件中:

```
import sys
from datetime import datetime
```

```
from supervisor import childutils # 通过childutils模块来创建事件监听

def write_log(headers, payload):
    if not headers['eventname'].startswith('PROCESS_STATE_'):
        return
    f = open('/tmp/log.txt', 'a')
    f.write(str(headers) + '\n\n') # 为了查看事件通知的格式
    pheaders, pdata = childutils.eventdata(payload + '\n')

    pheaders['dt'] = datetime.now() # 我们对事件的头信息做了扩充, 添加当前时间

    msg = ('[{dt}]Process {processname} in group {groupname} exited '
          'unexpectedly (pid {pid}) from state {from_state}\n').format(
        **pheaders)
    f.write(msg) # 当出现符合条件的三种事件, 会记录如上日志
    f.flush()
    f.close()

def main():
    while 1:
        headers, payload = childutils.listener.wait(sys.stdin, sys.stdout)
        write_log(headers, payload)
        childutils.listener.ok(sys.stdout)

if __name__ == '__main__':
    main()
```



Supervisor 有官方的插件集合 Superlance (<https://github.com/Supervisor/superlance>), 提供监控内存的使用、发送报警邮件等功能, 但是建议只把它用作自定义脚本实现时的参考。更多第三方应用和插件可以在 Third Party Applications and Libraries (<http://supervisord.org/plugins.html>) 找到。

使用 Supervisor

首先启动 Supervisord 进程:

```
> supervisord -c chapter7/section1/supervisord.conf
```


进程会以 daemon 的方式后台运行。使用 Supervisorctl 连接它：

```
> supervisorctl -c chapter7/section1/supervisord.conf
app_chapter6          RUNNING    pid 12316, uptime 0:00:11
listener              RUNNING    pid 12313, uptime 0:00:11
memcached:11212       RUNNING    pid 12314, uptime 0:00:11
memcached:11213       RUNNING    pid 12315, uptime 0:00:11
web_develop>
```

可以看到启动了两个 Memcached 进程、listener 进程和应用（Gunicorn）进程。现在管理进程：

```
web_develop> stop memcached:11212 # 关闭memcached:11212进程
memcached:11212: stopped
web_develop> status
app_chapter6          STARTING
listener              RUNNING    pid 17604, uptime 0:02:15
memcached:11212       STOPPED    Jun 04 01:21 AM
memcached:11213       RUNNING    pid 17606, uptime 0:02:15
web_develop> update # 更新supervisord.conf配置后，可以使用这个命令让配置生效
web_develop> start memcached:11212 # 重新启动
memcached:11212: started
web_develop> pid app_chapter6 # 查看进程id
12316
web_develop> maintail # 查看Supervisor日志
web_develop> tail -10 listener # 查看listener进程日志，支持进程名自动补全
2
OKREADY
```

通过 help 命令可获取全部命令的列表。管理命令可以作为 supervisorctl 的参数来使用：

```
> supervisorctl -c chapter7/section1/supervisord.conf stop memcached:11212
```

由于监听程序会监听 PROCESS_STATE_STOPPED，所以会记录停止 memcached:11212 的事件。查看/tmp/log.txt，就会看到如下记录：

```
{'ver': '3.0', 'poolserial': '3', 'len': '67', 'server': 'supervisor', 'eventname': 'PROCESS_STATE_STOPPED', 'serial': '3', 'pool': 'listener'}
```

```
[2016-06-04 01:21:11.914393]Process 11212 in group memcached exited unexpectedly (pid 12314) from state STOPPING
```

还可以通过事件返回的结果来组织内容发送邮件、HTTP 请求、短信，甚至触发某些其他操作。

可以访问 <http://localhost:5000>，看到 HTTP 的管理页面。因为在 [inet_http_server] 块中配置了用户和密码，所以访问的时候是有用户访问验证的。

除此之外，还可以通过 XML_RPC 接口来管理进程：

```
In : import xmlrpclib
In : server = xmlrpclib.Server('http://dongwm:123@localhost:5000/RPC2')
In : server.supervisor.stopProcess('memcached:11212')
Out: True
In : server.supervisor.getProcessInfo('memcached:11212')
Out:
{'description': 'Jun 04 01:31 AM',
 'exitstatus': 0,
 'group': 'memcached',
 'logfile': '/var/log/supervisord/memcached.log',
 'name': '11212',
 'now': 1465003888,
 'pid': 0,
 'spawnerr': '',
 'start': 1465003311,
 'state': 0,
 'statename': 'STOPPED',
 'stderr_logfile': '',
 'stdout_logfile': '/var/log/supervisord/memcached.log',
 'stop': 1465003867}

In : server.supervisor.startProcess('memcached:11212')
Out: True
```



支持的全部命令可以使用 `server.system.listMethods()` 获取。

之前的例子是在虚拟环境中实现的，如果 Supervisor 安装在全局而要使用虚拟环境，可以通过如下两种方法。

1. 把 program 项中的 command 改成完整路径：

```
[program:app_chapter6]
command=%(ENV_HOME)s/.virtualenvs/venv/bin/gunicorn -w 3 chapter6.section1.app:
app -b 0.0.0.0:9000
```

2. 使用参数 environment 把全局环境目录放进 PATH：

```
[program:app_chapter6]
environment = PATH=%(ENV_HOME)s/.virtualenvs/venv/bin:%(ENV_PATH)s
command=gunicorn -w 3 chapter6.section1.app:app -b 0.0.0.0:9000
```

除了使用 Supervisor 管理进程，还可以选择 Upstart 和 Systemd 来管理，推荐 Systemd，因为它的设计更优秀，越来越多的 Linux 发行版已经选择它。

应用部署 Fabric

当服务器规模比较小时，部署应用通常需要使用 SSH 登录服务器，备份，执行部署命令，退出，再登录另外一台服务器执行上述操作。这个过程会花费很多时间做一些重复性的工作，还需要确保执行正确。Fabric 正是为了提高这些基于 SSH 的应用部署和系统管理效率而诞生的。它通过原生的 SSH 库 Paramiko（库的作者就是 Fabric 的作者）实现与远程服务器的自动化交互，直接用命令行就可以在远程服务器上执行任务。

我们先安装它：

```
> pip install fabric
```

看一个非常简单的例子（fabfile.py）：

```
from fabric.api import run
```

```
def hostname():  
    run('hostname')
```

```
def ls(path='.'):  
    local('ls {}'.format(path))
```

执行一下：

```
> fab -H localhost,192.168.0.130 -f chapter7/section2/fabfile.py hostname  
[localhost] Executing task 'hostname'  
[localhost] run: hostname  
[localhost] Login password for 'vagrant':  
[localhost] out: WEB  
[localhost] out:  
  
[192.168.0.130] Executing task 'hostname'  
[192.168.0.130] run: hostname  
[192.168.0.130] out: CODE  
[192.168.0.130] out:
```

```
Done.  
Disconnecting from 192.168.0.130... done.  
Disconnecting from localhost... done.
```

上面的 hostname 函数没有参数，但是函数 ls 可以接受参数。带参数的函数通常使用默认参数：

```
> fab -H 192.168.0.132 -f chapter7/section2/fabfile.py ls # 通过默认值列出当前目录下的内容
> fab -H 192.168.0.132 -f chapter7/section2/fabfile.py ls:/home # 列出/home目录下的内容
> fab -H 192.168.0.132 -f chapter7/section2/fabfile.py ls:path=/home # 和上面的等价, 使用更明确地语法
> cd chapter7/section2
> fab -H 192.168.0.132 ls # 如果切换到fabfile.py的父级目录, 就不需要使用-f参数了
```

上述任务在执行时需要输入密码, 通常需要添加 SSH 信任, 添加之后不需要密码就能登录了:

```
> ssh-keygen
> ssh-copy-id -i ~/.ssh/id_rsa.pub localhost
> ssh-copy-id -i ~/.ssh/id_rsa.pub 192.168.0.130
```

SSH 信任是把双刃剑, 一定要注意服务器的安全, 尤其是双向的 SSH 信任。

Fabric 应用接口

Fabric 提供非常多的接口。基于封装、便捷性等考虑, 这些接口都放在 `fabric.api` 中来维护。主要分为如下 6 种接口。

操作

1. `run`: 用来在远程服务器上执行 shell 命令。

```
run('uptime')
result = run('ls -l /var/www')
print result.failed # 可以把执行的结果保存下来
```

2. `sudo`: 使用超级用户 (或者其他用户) 权限在远程服务器上执行 shell 命令。

```
sudo('ls /tmp/')
sudo('mkdir /data/nginx/tmp', user='nginx')
sudo('ls /data', user=1001)
result = sudo('ls /tmp/')
```

3. `require`: 检查环境变量中是否有 `require` 中定义的变量, 没有的话会直接终止执行。
4. `reboot`: 重新启动远程服务器。
5. `prompt`: 交互提示, 作用类似 `raw_input`。

```
username = prompt('Please specify username: ')
prompt('Specify favorite choice: ', 'choice', default=1, validate=int) # 输入的结果存在env.choice里面, 默认值为1, 会验证这个值是否是int
```

- 6. `get`: 从远程服务器下载文件。
- 7. `put`: 向远程服务器上传文件。
- 8. `open_shell`: 在远程服务器行启动一个完备的 `shell`。
- 9. `local`: 执行本地命令。

上下文管理

它们是一系列需使用 `with` 语法的函数。

- 1. `cd`: 切换远程目录。

```
with cd('/var/www'):  
    run('ls') # 相当于 cd /var/www && ls
```
- 2. `lcd`: 作用同 `cd`, 但是切换的是本地目录。
- 3. `char_buffered`: 强制本地终端的管道类型是 `character` 而不是 `line` 和 `buffered`。
- 4. `hide`: 隐藏指定类型的输出。

```
with hide('running', 'stdout', 'stderr'):  
    run('ls /var/www')
```

`hide` 的可选类型有 7 种, 如表 7.1 所示。

表 7.1 `hide` 的可选输出类型及其含义

类 型	含 义
<code>status</code>	状态信息, 比如服务器断开了连接, 用户使用了 <code>Ctrl-C</code> 等
<code>aborts</code>	中止信息, 一般在把 <code>Fabric</code> 当作库的时候需要关闭这样的状态信息
<code>warnings</code>	警告信息, 如果错误是预期的, 警告信息就会关闭
<code>running</code>	运行过程中的一些输出
<code>stdout</code>	本地或者远程与命令相关的非错误输出
<code>stderr</code>	本地或者远程与命令相关的错误输出
<code>user</code>	用户创建的输出

- 5. `path`: 增加环境变量 `PATH` 的路径。
- 6. `prefix`: 对于 `prefix` 下的每个命令都整体执行一遍。

```
with prefix('echo 1'):  
    run('echo 2')  
    run('echo 3')
```

相当于：

```
echo 1 && echo 2
echo 1 && echo 3
```

7. `quiet`：相当于隐藏了全部信息，在执行错误的时候仅仅发出警告信息。

8. `remote_tunnel`：通过 SSH 的端口转发建立转发通道。

```
with remote_tunnel(3306):
    run('mysql -u root -p password')
```

这样就在本地连接了远程的 MySQL 实例。

9. `settings`：临时覆盖 `env` 变量。

10. `shell_env`：设置 `shell` 环境变量。

11. `show`：和 `hide` 相反，显示指定类型的输出。

12. `warn_only`：`settings(warn_only=True)` 的别名。

装饰器

通过装饰器可以更方便地操作 `Fabric`。

1. `hosts`：定义会执行此函数的远程服务器列表。

```
@hosts('user1@host1', 'host2', 'user2@host3')
def my_func():
    pass
```

2. `parallel`：使用并行执行。

```
@parallel
def runs_in_parallel():
    pass
```

3. `roles`：对主机按角色分组，定义执行函数的角色。

```
env.roledefs.update({
    'webserver': ['www1', 'www2'],
    'dbserver': ['db1']
})
```

```
@roles('webserver', 'dbserver')
def my_func():
    pass
```

4. `runs_once`：只执行一次，防止函数被多次运行。

5. serial: 强制远程服务器使用串行执行。
6. task: 定义任务的函数。
7. with_settings: 是 fabric.api.settings 的装饰器版本。

功能函数

1. abort: 终止函数执行, 打印错误信息到 stderr, 使用 1 作为退出码。
2. warn: 输出警告信息, 但是不会终止函数的执行。
3. puts: 打印输出。

状态

1. env: 当前环境的字典。我们可以在部署程序中直接指定环境的值。
2. output: 当前输出类型的字典。

任务执行。

可以通过 execute 函数在运行期执行任务函数:

```
from fabric.api import task, execute, run
```

```
@task
def info():
    return run("print info")
```

```
@task
def go():
    results = execute(info)
    print results
```

使用 Fabric 管理 Flask 应用

本节我们来实现一个部署 Flask 应用的例子, 它实现如下两个功能:

- 同时查看全部服务器上的根据内存占用排序的进程。
- 通过拷贝文件, 杀掉原来的应用进程, 最后启动新应用来实现部署。同时检查远程服务器上是否安装了 Gunicorn, 如果没有安装的话, 就自动安装 (fabfile_app.py)。

```
from fabric.api import (
```

```
cd, run, task, env, roles, put, execute, parallel, hide, settings, sudo)
from fabric.colors import red, green
from fabric.contrib.files import exists

env.roldefs.update({
    'webserver': ['192.168.0.130', '192.168.0.132'],
    'dbserver': ['192.168.0.175']
})

@task
@parallel(pool_size=5)
@roles('webserver', 'dbserver')
def top_mem_proc():
    run('ps -ef |sort -rk4 |head')

@task
@roles('webserver')
def upload():
    put('chapter6/section1/run.py', '/tmp/run.py')

def check_command(cmd):
    rs = run('command -v {} >/dev/null 2>&1'.format(cmd))
    return rs.return_code == 0

@task
def install_it(package):
    sudo('pip install {}'.format(package))
    print green('{} installed'.format(package)) # 输出颜色为绿色

@task
@roles('webserver')
def restart_app():
    with cd('/tmp'):
        if exists('/tmp/app.pid'):
            pid = run('cat /tmp/app.pid')
            run('kill -9 {}; rm /tmp/app.pid'.format(pid))
        else:
            print red('pid file not exists!') # 错误提示为红色
    with settings(hide('everything'), warn_only=True):
        if not check_command('gunicorn'):
            install_it('gunicorn')
```



```
with hide('running', 'stdout'):
    run('gunicorn -w 3 run:app -b 0.0.0.0:8000 -D -p /tmp/app.pid --log-file /
        tmp/app.log') # noqa
```

```
@task
def deploy():
    execute(upload)
    execute(restart_app)
```

现在就可以部署应用了：

```
> fab -f chapter7/section2/fabfile_app.py deploy
```



使用 put 在处理多文件时很低效，可以使用 rsync_project 或者 upload_project 的方式对整个项目进行部署。

```
from fabric.contrib.project import rsync_project

rsync_project(
    remote_dir='/opt/app',
    local_dir='.',
    exclude=('*_local.py', '*.pyc')
)
```

配置管理工具 SaltStack 和 Ansible

系统管理员经常陷入一系列重复的任务中，如新服务器初始化、升级软件包、远程执行、配置变更、维护定时任务、应用部署、重启服务、上线、回滚等。在配置工具没有出现之前，我们会维护大量用于不同目的脚本，甚至组合出不同的流程。在服务器规模比较小时，配置管理的重要性并不突出，但是当要维护成千上万的服务器时，就会发现如下几点问题：

1. 无法保证快速上线。如果没有自动化批量部署方案，对全部服务器的部署过程耗时很长，工作量也是巨大的。
2. 跨平台支持有限。往往不同的平台重新实现一遍操作代码。
3. 可维护性差。随着需要维护的系统架构越来越复杂，如果没有使用良好的模块化设计，会造成任务程序越来越复杂，而且常常由于没有社区的贡献，架构灵活度也很差，这不可避免地让维护者选择重新造轮子。

配置管理工具有两个 Python 的实现：Saltstack 和 Ansible，配置管理的特点就是通过配置而不是脚本来管理远程服务器，无论是单台服务器还是上万台服务器，通过配置来管理都很方便。

SaltStack

SaltStack 和 Puppet 一样采用 C/S 架构，但相对 Puppet 而言，它更轻量，配置语法使用 YML，其发布订阅系统是基于 ZeroMQ 的，以广播的形式通知客户端（官方称为 Minion）。它封装了常用的命令执行模块，包括管理软件包、管理用户、传输文件、数据库管理等等。在功能上，SaltStack 大概相当于 Fabric + Puppet。

我们先在 Minion/Server 端安装对应的软件包：

```
> curl -L https://bootstrap.saltstack.com -o install_salt.sh
# Server端使用如下命令：
> sudo sh install_salt.sh -P -M
# Minion端使用如下命令：
> sudo sh install_salt.sh -P
```

首先修改 Master 端的配置，把/etc/salt/master 里面的 hash_type 改成“sha256”，重启 Master：

```
> sudo /etc/init.d/salt-master restart
```

查看 Master 的识别指纹（Fingerprint）：

```
> sudo salt-key -F master|grep master.pub
master.pub: 67:34:6c:aa:43:ff:36:ca:41:b5:fa:ee:37:11:cc:44:71:49:ce:3f:8b:27:1d:06:9e:
a5:9d:aa:b6:bf:45:11
```

然后对每个 Minion 端修改配置，可以把/etc/salt/minion 里面的如下代码行改掉：

```
master: 192.168.0.175 # Master的IP，注意冒号之后要有一个空格
master_finger: '67:34:6c:aa:43:ff:36:ca:41:b5:fa:ee:37:11:cc:44:71:49:ce:3f:8b:27:1d:
:06:9e:a5:9d:aa:b6:bf:45:11'
id: 130 # 设定客户端的Minion id标识，Salt默认使用socket.getfqdn()的结果作为标识。
不同的Minion端需要使用不同的id
```

重启 Minion 端：

```
> sudo /etc/init.d/salt-minion restart
```

这个时候在 Master 上可以看到这个 Minion 端的 key：

```
> sudo salt-key -L
Accepted Keys:
Denied Keys:
```

Unaccepted Keys:

130

Rejected Keys:

接受它:

```
> sudo salt-key -a 130 # 可以使用-A接受全部
```

通过 ping 测试连通性:

```
> sudo salt '*' test.ping # 支持通配符, 星号表示所有客户端
```

130:

True

现在对 Minion 端进行一系列操作:

```
> sudo salt 130 cmd.run 'hostname' # 获取主机名
```

```
> sudo salt 130 cmd.run "ifconfig|grep -1 eth1|grep -Eo 'inet (addr:)?([0-9]*\.){3}[0-9]*'|cut -d ':' -f 2" # 获取IP地址
```

```
> sudo salt 130 disk.usage # 使用内建的方法获得硬盘使用情况。全部的内建模块可以在https://docs.saltstack.com/en/latest/ref/modules/all/index.html找到, 不同的模块可能还需要安装对应的依赖, 否则不能使用
```

```
> sudo salt '*' pip.install markdown,psutil # 给客户端统一通过pip安装markdown和psutil
```

```
> sudo salt '*' pip.install pyOpenSSL upgrade=True # 通过参数的方式实现 pip install -U
```

```
> sudo salt 130 ps.kill_pid 111 signal=9 # ps模块是psutil模块的缩写
```

```
> sudo salt 130 ps.top 5 10
```



如果执行过程中没有符合预期的结果, 可以使用“`sudo salt-minion -l debug`”或者“`sudo salt-master -l debug`”的方式启动, 就可以通过详细的 Debug 输出了解到问题出在什么地方了。

使用 Grains

上面我们是通过匹配 Minion id (也包含通配符、正则匹配等方式) 选择目标服务器的, 还可以通过 Salt 的 Grains 特性来选择。Grains 是每一个 Minion 端自身的静态属性, 以 Python 字典的形式存放在 Minion 端:

```
> sudo salt 130 grains.items |grep -A 1 'os:' # 静态属性非常多, 我们只过滤操作系统这一项。只选择os也可以使用sudo salt '*' grains.item os
```

os:

Ubuntu # 130是一个Ubuntu服务器

可以通过操作系统来选择目标服务器：

```
> sudo salt -G 'os:Ubuntu' test.ping
```

全部的 Grains 名字可以通过如下命令获取：

```
> sudo salt '*' grains.ls
```

除此之外，我们还可以选择使用复合匹配（Compound matchers）：

```
> sudo salt -C 'E@13* and G@os:Ubuntu or webserv*' test.ping
```

其中 E 表示正则匹配，G 表示 Grains。可以用布尔操作符连接多个目标条件来获取目录服务器。



Salt 支持批量执行，在某些工作中非常有用，如重启负载均衡集群中的 Web 服务：

```
> sudo salt '*' -b 10 test.ping # 同一时间只有10台机器运行此命令，当
    有Minion返回执行结果时，再让下一个Minion执行，直到所有目标机器执
    行完成
> sudo salt -G 'os:Ubuntu' -b 10% test.ping # 同一时间只有10%的服务器运
    行
```

配置状态

和 Puppet 一样，在 Salt 中可以描述系统的目标状态。Salt 将其称为一个 State，Salt 的 State 模块文件用 YAML 写成，以 .sls 结尾（SaLt State 的大写字母的缩写）。它们从功能上等同于 Puppet 模块的 manifest 文件，只是后者用 Puppet DSL 写成，以 .pp 结尾。

/etc/salt/master 中的 file_roots 指定了定义 Salt 状态的文件的目录，我们把它修改成如下内容：

file_roots:

```
base: # base是必须存在的，其他的子项都是可选存在的
  - /srv/salt
dev: # 开发环境的目录，我通常习惯先放在dev目录下，在测试环境试验问题后再合并
    到prod中
  - /srv/salt/dev
prod: # 生产环境的目录
  - /srv/salt/prod
```

将本节全部源文件拷贝到 /srv 下：

```
> sudo chown ubuntu:ubuntu /srv -R
> cp chapter7/section3/* /srv -r
```

一般而言,对某个软件进行配置管理应该创建一个同名的目录,在目录下创建 `init.sls` 文件。如果配置很简单,也可以直接使用同名的 `.sls` 文件。现在演示对 Redis 进行配置管理,添加 `/srv/salt/dev/redis/init.sls` 作为入口,入口文件一般没有具体的逻辑,只是通过 `include` 包含一些状态文件:

```
> cat /srv/salt/dev/redis/init.sls
include:
  - redis.server
```



当同时存在 `redis.sls` 和 `redis/init.sls` 时,将指向 `redis.sls`,忽略 `redis/init.sls`。

再看看目录下的其他文件。

1. `common.sls`: 安装 Redis。

```
{% from "./redis/map.jinja" import redis_settings with context %} # Jinja2语
法, 引用其他Jinja2文件
install-redis:
  pkg.installed: #pkg是自带的状态模块。全部状态模块可以在https://docs.saltstack.com/en/latest/ref/states/all/找到。installed是确保指定的包被
  安装。
  - name: {{ redis_settings.pkg_name }}
  {% if redis_settings.version is defined %}
  - version: {{ redis_settings.version }}
  {% endif %}
```

2. `server.sls`: 配置 Redis。

```
include:
  - redis.common

{% from "redis/map.jinja" import redis_settings with context %}

{% set cfg_version    = redis_settings.cfg_version -%}
{% set cfg_name       = redis_settings.cfg_name -%}
{% set pkg_name       = redis_settings.pkg_name -%}

redis_config:
  file.managed: # file也是一个自带的状态模块, managed表示可以从Master端下载
    文件, 这个文件可能需要先使用对应的模板渲染。
  - name: {{ cfg_name }}
  - template: jinja
  - source: salt://redis/files/redis-{{ cfg_version }}.conf.jinja
```

```
- require:
  - pkg: {{ pkg_name }}
```

```
redis_service:
  service.{{ redis_settings.svc_state }}: # 默认是service.running, 也就是验证
    服务在运行中
  - name: {{ redis_settings.svc_name }}
  - enable: {{ redis_settings.svc_onboot }}
  - watch: # 表示里面任何一个状态变化就触发
    - file: {{ cfg_name }}
  - require: # 在执行这一步之前需要满足的东西都列在下面; 不满足就不执行
    - pkg: {{ pkg_name }}
```

3. files/redis-3.0.conf.jinja: Redis 3.0 配置的模板, files 目录下还包含 redis-2.8.conf.jinja, 对应 Redis 2.8 的配置。

```
{% from "redis/map.jinja" import redis_settings with context %}
daemonize {{ redis_settings.daemonize }}

pidfile {{ redis_settings.pidfile }}
port {{ redis_settings.port }}
...
```

4. map.jinja: 包含 Redis 的公用默认设置以及不同操作系统下的配置的映射。

在部署的时候 Salt 会先通过 Jinja2 的模板替换生成 Redis 配置文件, 再复制文件到目标服务器。

使用 Pillar

Pillar 用来存放键值对变量, 变量存放在 Master 端, 由 Master 编译好后下发给 Minion 端。它通常用来存放敏感数据 (如 ssh key、加密证书等) 和一些通用的、有平台差异性的变量。定义 Pillar 变量的文件目录是通过/etc/salt/master 中的 pillar_roots 指定的, 我们把它修改成如下内容:

```
> grep -A 3 "pillar_roots:" /etc/salt/master|grep -v "^#"
pillar_roots:
  base:
    - /srv/pillar
```

与状态文件相似, Pillar 也有 top.sls, 使用相同的匹配方式将数据应用到 Minion 端上:

```
base:
  '*':
```

```
- packages # 所有Minion端都要加载
'13*': # 13开头的Minion端需要加载redis这个配置
- redis
```

不同系统之间的包的名字不同，可以在 packages.sls 里面统一配置：

```
{% if grains['os'] == 'RedHat' %}
  python-dev: python-devel
  git: git
{% elif grains['os'] in ('Debian', 'Ubuntu') %}
  python-dev: python-dev
  git: git-core
{% endif %}
```

Redis 的基本设置存放在 redis.sls 中：

```
redis:
  root_dir: /var/lib/redis
  user: redis
  port: 6379
  bind: 127.0.0.1
  snapshots:
    - '900 1'
    - '300 10'
    - '60 10000'

  lookup:
    svc_state: running
    cfg_name: /etc/redis.conf
    pkg_name: redis-server
    svc_name: redis-server
```

现在可以通过 Pillar 的键过滤目标服务器了：

```
> sudo salt -I 'redis:port:6379' test.ping
```

Salt 模板

现在 Salt 支持 Jinja2、Mako、Wempy 三种模板语言和 YAML、JSON 两种格式的搭配，本节选用的是 Jinja + YAML。如果使用非默认的组合，可以在开头添加一个 Shebang，比如“#!jinja|mako|yaml”表示使用 Jinja2 + Mako + YAML 的组合。

现在看一下之前提到的模板文件 map.jinja：

```
> cat /srv/salt/dev/redis/map.jinja
{% set default_settings = {
```

```

    'redis': {
        'bind': '127.0.0.1',
        'database_count': 16,
        'root_dir': '/var/lib/redis',
        'daemonize': 'yes',
        ... # 为节省篇幅, 省略大部分默认的设置
        'user': 'redis'
    }
}%}

{% set redis_settings = salt['pillar.get']('redis', default=default_settings.redis,
merge=True) %}

```

redis_settings 最后会从 Pillar 里获得对应的字段值做替换。

现在就可以安装 Redis 了:

```
> sudo salt '*' state.highstate
```

当 Master 运行 state.highstate, Minion 端会先下载 top.sls, 如果发现有符合它的匹配, 就下载相应的文件, 编译并运行。运行完成后, Minion 端会返回结果的汇总。

SSH 模式

使用 SSH 模式不需要安装 Minion 端就可以实现远程部署, 我们需要先安装 salt-ssh:

```
> sudo apt-get install salt-ssh -y
```

现在给 192.168.0.132 添加 SSH 信任:

```
> sudo ssh-copy-id -i /etc/salt/pki/master/ssh/salt-ssh.rsa.pub vagrant@192.168.0.132
```

定义文件:

```

> cat /etc/salt/roster
web1:
    host: 192.168.0.132
    user: ubuntu
web2:
    host: 192.168.0.133
    user: ubuntu
web3:
    host: 192.168.0.221
    user: vagrant
    passwd: vagrant # 不加SSH, 用密码也可以
    sudo: True # 是否需要sudo, 默认是False

```


之前的/srv/salt/top.sls 使用 IP 作为标识，现在使用以 web 开头的标识。需要修改一下：

```
> cat /srv/salt/top.sls
dev:
  'web*':
    - redis
```

现在就可以一样地使用命令了：

```
> sudo salt-ssh '*' test.ping
> sudo salt-ssh web3 state.highstate # 为了演示，web1和web2没有设置sudo权限，不能操作/etc下的Redis配置。所以只能部署web3
```

Ansible

Ansible 也是配置管理和应用部署工具，由 Michael DeHaan 创建，他同时也是知名软件 Cobbler 和 Func 的作者。Ansible 默认通过 SSH 协议管理服务器，不需要安装客户端。

我们先安装它：

```
> pip install ansible
> python -c 'import ansible; print ansible.__version__'
2.1.0.0 # 2.0的版本做了一次较大的重构，并重新组织了代码
```

定义服务器的清单列表文件：

```
> sudo mkdir /etc/ansible
> cat /etc/ansible/hosts # 当然也可以使用其他名字，通过-i指定的方式使用
[webservers]
192.168.0.132
192.168.0.174
192.168.0.101
```

webservers 这个组有三个 IP。还可以通过如下方式表示复杂的服务器类型：

```
[dbservers]
db[01:20].dongwm.com # 也就是db01到db20这20个服务器
c.dongwm.com:5501 # 使用5501作为ssh端口
db21 ansible_ssh_port=5555 ansible_ssh_host=192.168.0.100 # 设置别名和端口
localhost ansible_connection=local # 设置本地的连接类型
192.168.0.201 ansible_connection=ssh ansible_ssh_user=dongwm
```

现在执行远程命令（省略实现 SSH 信任的步骤）：

```
> ansible webservers -a "whoami"
> ansible all -m ping # ping是它内置的模块。全部的模块可以在http://docs.ansible.com/ansible/list\_of\_all\_modules.html找到
```

```

> ansible webserver -m service -a "name=nginx state=restarted" --become # 重启Nginx服务, sudo命令未来会被废弃, 使用become替代
> ansible webserver -a "hostname" -f 3 # -f表示同时执行, 提高并发效率
> ansible webserver -m file -a "dest=/tmp/file mode=600 owner=root group=root" --become # 可以修改目标文件权限、用户组等
> echo 1 >> /srv/ansible/file
> ansible webserver -m copy -a "src=/srv/ansible/file dest=/tmp/file" # 把文件复制到目标服务器的/tmp/目录下
> ansible webserver -m apt -a "name=nginx state=present" --become # 确保安装, 但是不更新。如果使用“state=latest”表示确保安装到最新版; “state=absent”表示确保没有安装。不同的系统安装包的模块不同, apt模块仅限Debian/Ubuntu系统
> ansible 192.168.0.132 -m git -a "repo=https://github.com/Icinga/icinga2.git dest=/opt/icinga version=HEAD" # 使用Git克隆最新的Icinga2代码到192.168.0.132的/opt/icinga目录

```

Playbooks

剧本 (Playbooks) 是 Ansible 的配置、部署、编排语言。它们可以被描述为一个需要远程主机执行的命令集合。

本节我们使用剧本实现一个 Redis 部署的功能。Playbooks 使用 YAML 语法, 看一下主文件 main.yml:

```

---
- hosts: webserver
  remote_user: ubuntu
  become: yes
  become_method: sudo
  vars_files: # 用到的变量单独存放在YAML文件中
    - vars.yml
  tasks: # 任务列表, 按顺序执行
    - name: ensure packages installed
      apt: pkg={{ item }} state=latest
      with_items:
        - make
        - build-essential
        - tcl8.5

    - name: ensure vm.overcommit_memory equal 1 # 通常应该在新服务器上架的初始化过程中设置内核参数
      sysctl: name="vm.overcommit_memory" value=1 sysctl_set=yes state=present reload=yes

    - name: download latest stable redis
      get_url: url=http://download.redis.io/redis-stable.tar.gz dest=/tmp/redis-stable.

```

```

tar.gz

- name: untar redis
  unarchive: src=/tmp/redis-stable.tar.gz dest=/tmp

- name: build redis
  command: make -C /tmp/redis-stable

- name: create redis group
  group: name=redis state=present system=yes

- name: create redis user
  user: name=redis group=redis createhome=no shell=/bin/false system=yes state=
    present

- name: make sure that /etc/redis exists
  file: path=/etc/redis state=directory mode=0755

- name: make sure that /var/db/redis exists
  file: path=/var/db/redis state=directory mode=0755 group=redis owner=redis

- name: copy service file
  copy: src=templates/redis-server.service dest=/lib/systemd/system/redis-server.
    service # 可以被systemd识别，就能执行“sudo systemctl start redis-server
    ”了

- name: installing redis binaries
  command: cp /tmp/redis-stable/src/{{ item }} /usr/local/bin
  with_items: # 这个列表会依次执行上面的命令，item就是每次要执行命令的文件名
    字
    - redis-server
    - redis-cli
    - redis-check-aof

- name: copy redis.conf file
  template: src=templates/redis.conf.j2 dest=/etc/redis/redis.conf group=redis
    owner=redis # 需要使用template这个模块进行Jinja2渲染
  notify:
    - restart redis # 通知触发的命令在handlers中需要定义

- name: cleaning up build files
  command: rm -rf /tmp/{{ item }}
  with_items:
    - redis-stable
    - redis-stable.tar.gz

```

```
handlers: # handlers中定义了上面notify中触发的命令具体内容
- name: restart redis
  service: name=redis state=restarted # service指service模块
```

变量集中在 vars.yml 文件中，和之前的 Jinja2 中的默认设置值一样，这里也只展示一小部分：

```
> cat vars.yml
---
redis_settings:
  appendfilename: appendonly.aof
  appendonly: 'no' # no和yes需要使用加引号的字符串，否则会被当成True/False
  appendfsync: everysec
  auto_aof_rewrite_min_size: 64mb
  auto_aof_rewrite_percentage: 100
...
```



Ubuntu 16.04 LTS 已经使用 systemd 作为系统和服务管理器，redis-server.service 可以作为 Ubuntu 的系统服务的配置文件。不使用 systemd 的系统可以使用 upstart，首先把 upstart 脚本拷贝到/etc/init 目录下：

```
> sudo cp templates/upstart.conf /etc/init
```

然后启动服务：

```
> sudo service redis-server start
```

Redis 配置文件由前面介绍 Salt 的那一节里用到 redis-3.0.conf.jinja 模板生成，为了符合 Ansible 的习惯，模板改用了 j2 后缀。由于没有用到 map.jinja 文件，所以没有开始的 import 语句：

```
> cat /srv/ansible/dev/redis/templates/redis.conf.j2
daemonize {{ redis_settings.daemonize }}

pidfile {{ redis_settings.pidfile }}
port {{ redis_settings.port }}
...
```

现在就可以给 webservers 集群安装 Redis 了：

```
> ansible-playbook /srv/ansible/dev/redis/main.yml
```

更多的剧本示例可以参看 ansible-examples (<https://github.com/ansible/ansible-examples>)。

Role 和 Galaxy

需要部署的内容越来越多，脚本文件也会越来越多，这个时候势必出现很多重复的代码，最好的解决办法是使用 Role（角色）重新组织这些脚本文件，把它们分层，结构化地使用。一个大型的部署脚本目录结构如下（假设服务的名字叫作 xx）：

```
> tree /srv/ansible/dev/xx
├── defaults # 默认变量
│   └── main.yml # 存放成目录的方式后，main.yml就是主入口
├── handlers # 把事件处理独立成目录
│   └── main.yml
├── meta # 存放元数据和依赖的其他角色的信息
│   └── main.yml
├── files # 需要拷贝的文件
├── tasks # 把执行的任务独立成目录
│   ├── install.yml # 安装相关的设置
│   ├── local_facts.yml
│   ├── main.yml # 通过main.yml导入其他yaml的设置
│   └── configuration.yml # 配置相关的设置
├── templates # 模板
│   ├── Debian # 不同平台的模板存放的目录不同
│   │   └── xx.conf.j2 # 配置可能同时存在多个版本（一般是业务需要）。比如之前看
│   │       到的redis-3.8和redis-3.0
│   ├── default
│   │   └── xx.conf.j2
│   ├── RedHat
│   │   └── xx.conf.j2
│   ├── upstart.conf.j2
│   └── redis-server.service.j2
└── vars # 把变量存储在独立的目录中
    └── main.yml
```

tasks/main.yml 中使用如下方法导入其他 YAML 配置：

```
---
- include: install.yml
- include: configuration.yml
```

假设部署 webserver 的时候需要安装 xx 服务，可以这样编写脚本：

```
---
- hosts: webserver
  roles:
    - common # 假设/srv/ansible/dev/common有与xx同样的目录结构，它存放了相对通用的
              脚本
```

- xx

在部署 webserver 的时候，会对 roles 中定义的角色（假如叫作 a）做如下的处理：

- 如果 roles/a/tasks/main.yml 存在，将其加入到剧本。
- 如果 roles/a/handlers/main.yml 存在，将其加入到剧本。
- 如果 roles/a/vars/main.yml 存在，将其加入到剧本。
- 如果 roles/a/meta/main.yml 存在，将其加入到剧本。

Ansible Galaxy (<https://galaxy.ansible.com/>) 是官方提供的一个免费的搜索、查询、下载社区提供的角色（Role）服务。在上面可以找到非常多的剧本例子，甚至可以直接在自己的项目中使用。现在通过 Galaxy 安装 Jenkins（一个持续集成工具）的方法。

首先在主控端安装角色：

```
> cat /etc/ansible/ansible.cfg # 默认的角色会存放在/etc/ansible/roles目录下，由于我
    们使用虚拟环境，没有/etc目录下的写权限，需要自定义角色配置的存放目录
[defaults]
roles_path = /srv/ansible/roles
> ansible-galaxy install dongweiming.jenkins
```

配置角色：

```
> mkdir /srv/ansible/dev/jenkins/host_vars -p
> cat /srv/ansible/dev/jenkins/jenkins.host
[jenkins]
192.168.0.132
> cat /srv/ansible/dev/jenkins/host_vars/192.168.0.132 # 当目标服务器的IP是192
    .168.0.132时，将使用这个文件定义的变量，同样的还有group_vars。这种方式更灵活
---
plugins:
  - 'ldap'
  - 'github'
  - 'translation'
  - 'preSCMbuildstep'
port: 9000
prefix: '/build'
email:
  smtp_host: 'mail.dongwm.com'
  smtp_ssl: 'true'
  default_email_suffix: '@dongwm.com'
java_version: "openjdk-9"

> cat /srv/ansible/dev/jenkins/jenkins.yml
---
```

```
- hosts: jenkins
  become: yes
  roles:
    - dongweiming.jenkins
```

现在就可以给 192.168.0.132 这台服务器安装 Jenkins 了：

```
> cd /srv/ansible/dev/jenkins
> ansible-playbook -i jenkins.host jenkins.yml
```

使用 Psutil

Psutil 是一个跨平台的进程处理和系统工具模块，它可以获取系统 CPU、内存、硬盘和网络等信息，常用于系统资源的采集、监控、分析，以及进程管理。使用它可以用 Python 实现系统相关的命令行功能，比如 top、netstat、ps、iostat、ifconfig、df，具体实现可以参考项目代码下的 scripts 目录（<http://bit.ly/28SOjEX>）。目前使用它的应用有 Psdash、Glances 等。虽然它主要应用于运维开发中，但是在应用代码中也可以使用它增加一些这方面的功能。举两个例子：

- 启动定时任务时需要确保之前的定时任务都已完成，否则要直接退出或者杀掉这些之前没有完成的进程，再开始。
- 当应用在内存占用等方面有问题时，加入 Psutil 代码，监控对应内存（或其他指标）的使用情况来帮助确定原因。

我们先安装它：

```
> pip install psutil
```

常用的函数有如下 10 个。

1. cpu_percent：返回当前 CPU 的利用率的百分比。

```
In : psutil.cpu_percent() # 它接受两个参数：interval是统计的间隔，percpu为
    True会返回所有逻辑CPU的利用率的列表
Out: 32.6
```

2. cpu_count：获得逻辑（物理）CPU 的个数。

```
In : psutil.cpu_count()
Out: 24
In : psutil.cpu_count(logical=False) # 获取物理CPU数，会去掉超线程CPU的数量
Out: 2 # 数据差别很大
```

3. virtual_memory：内存占用情况。

```
In [2]: psutil.virtual_memory()
Out[2]: svmem(total=101540593664L, available=77059371008L, percent=24.1, used
         =89900863488L, free=11639730176L, active=25916870656, inactive=57613012992,
         buffers=6373376L, cached=65413267456)
```

其中:

- total 表示全部内存。
 - available 表示可以用来分配的内存。不同系统的计算方式不同, 在 Linux 下是 `free + buffers + cached`。
 - percent 表示使用的内存的占比, 也就是 $(total - available) / total * 100$ 。
 - used 表示已经用到的内存。不同系统的计算方式不同。
 - free 表示未被分配的内存。Linux 下不包括 buffers 和 cached。
4. `disk_partitions`: 硬盘分区信息, 返回所有挂载的分区的信息的列表。列表中的每一项类似于 `df` 命令的格式输出, 包括分区、挂载点、文件系统格式、挂载参数等。

```
In : psutil.disk_partitions()
Out:
[sdiskpart(device='/dev/root', mountpoint='/', fstype='xfs', opts='rw,noatime,
         nodirtime,attr2,delaylog,nobarrier,noquota'),
 sdiskpart(device='/dev/sda2', mountpoint='/data1', fstype='xfs', opts='rw,
         noatime,nodirtime,nobarrier')]
```

5. `disk_usage`: 返回硬盘, 分区或者目录的使用情况, 单位为字节。

```
In : psutil.disk_usage('/')
Out: sdiskusage(total=42928640000, used=40646586368, free=2282053632, percent
      =94.7)
```

6. `pids`: 获得当前运行的程序进程的 id 列表。

```
In : pids = psutil.pids()
```

7. `pid_exists`: 检查程序 id 是否存在, 可以理解为 “`pid in psutil.pids()`” 的快捷方式。

```
In : psutil.pid_exists(11111)
Out: False
In : psutil.pid_exists(pids[10])
Out: True
```

8. `Process`: 操作进程类。

```
In : p = psutil.Process(pid=pids[100])
In : datetime.datetime.fromtimestamp(p.create_time()).strftime("%Y-%m-%d %H:%M:%S")
Out: '2016-03-06 06:17:04'
```



```
In : p.as_dict(attrs=['pid', 'name', 'username'])
Out: {'name': 'tail', 'pid': 1414, 'username': 'dongwm'}

In : p.cpu_percent(interval=1)
Out: 1.0

In : p.memory_info()
pmem(rss=15491072, vms=84025344, shared=5206016, text=2555904, lib=0, data
     =9891840, dirty=0)

In : p2 = psutil.Process() # 获取当前进程实例
In : p2
Out: <psutil.Process(pid=341, name='ipython') at 37342416>
In : p2.cpu_affinity([5]) # 把进程绑到第6个CPU上
In : p2.open_files()
Out:
[popenfile(path='/home/ubuntu/.ipython/profile_default/history.sqlite', fd=3),
 popenfile(path='/home/ubuntu/.ipython/profile_default/history.sqlite', fd=4)]
In : p2.cmdline()
Out: ['/usr/bin/python2.7', '/home/ubuntu/web_develop/venv/bin/ipython']
In : p2.kill() # 杀掉了自己
[1] 341 killed /home/ubuntu/web_develop/venv/bin/ipython
```

使用 Sentry 收集错误信息

很多开发者都有这样不愉快的经历：代码在测试环境运行很正常，一到线上就收到用户的反馈和投诉。但是用户的环境各不相同又很复杂，很多错误用户很难描述，甚至开发者都无法复现。但是问题确实存在，怎么办？

在笔者之前的工作中，开发者没有线上服务器的登录权限，申请紧急调试多有不便，即便登录服务器还要熟悉且遵守一系列的运维流程，这些都增加了我们定位问题的难度和时间。笔者当时想了如下一些办法：

- 当代码出现问题登录服务器后，在对应的位置添加调试代码，模拟请求或者等待用户触发。然后进一步排查，直到找到问题的原因。
- 开发时，在认为可能会有问题的地方加入一些异常处理，将异常和它相关的上下文捕捉到日志中，当出现问题时帮助定位。
- 项目依赖了一些便于调试和更快找到问题的第三方或者自己实现的模块，例如 `q` (<https://github.com/zestyping/q>)，它是 PyCon US 2013 的一个主题演讲 (<http://bit.ly/28Sbq6f>) 提到的模块，它会把执行的函数或者类的参数存在一个独立的日志里面。

这些确实够用了，但是还有 3 个问题：

1. 项目代码中出现一些带有调试意义的代码，既增加了代码量，也会让新的维护者感到疑惑。
2. 这种手段完全靠开发者对于项目的熟悉程度和解决问题，其不可预知性和效率可想而知。
3. 出现的异常都在时刻影响用户，需要第一时间修复解决，对于开发者的压力非常大，也扰乱了他们的日常工作。

Sentry 是一个基于 Django 的实时事件日志和聚合平台，于 2010 年诞生于 Disqus，它可以帮助我们程序的所有异常自动记录下来，在 Sentry 自带的用户页面上呈现并支持搜索等功能。其事件监控功能却不局限于 Python，对 Node.js、Go、Java 等语言的项目都可以做到无缝集成，它还能对 iOS、Android 移动客户端以及 Web 前端异常进行跟踪。处理异常是所有应用的必要工作，可以说 Sentry 是 Web 应用必备组件之一。它可以做如下事情：

- 系统发生异常时，可以通过 Sentry 查看到具体错误信息和触发错误的上下文，还能很容易地重现异常。
- 可以通过邮件等方式第一时间告知开发者出现了异常。
- 帮助我们了解系统都产生了哪些异常、异常发生的频率、开始时间、发生的频次等。
- 可以对系统异常分组，提供高级搜索等功能。
- 可以把它当成一个异常处理的在线讨论、反馈和事件跟踪的平台。

安装配置 Sentry

Sentry 依赖 PostgreSQL 等（Nginx、Redis 已安装过）软件包，需要先安装它们：

```
> echo "deb http://apt.postgresql.org/pub/repos/apt/ trusty-pgdg main" | sudo tee /etc/
apt/sources.list.d/postgresql.list
> wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key
add -
> sudo apt-get update
> sudo apt-get install python-dev libxslt1-dev libxml2-dev libz-dev libffi-dev libssl-
dev libpq-dev libyaml-dev postgresql-9.3 -yq
> sudo /etc/init.d/postgresql start
```

安装 Sentry：

```
> pip install sentry
> python -c 'import sentry; print sentry.__version__'
8.5.0 # Sentry 8使用React对前端进行了组件化重构，大部分的页面不再要求页面刷新，
      这提高了页面的访问效率
```

生成配置文件:

```
> sentry init
```

由于数据迁移的问题, Sentry 8 官方不再支持 MySQL 数据库, 唯一支持的生产环境数据库是 PostgreSQL。先配置 PostgreSQL 数据库:

```
> sudo su - postgres
$ psql
postgres=# CREATE USER ubuntu WITH PASSWORD '123';
CREATE ROLE
postgres=# CREATE DATABASE sentry OWNER ubuntu;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON DATABASE sentry to ubuntu;
GRANT
postgres=# \q
```

修改 ~/.sentry/sentry.conf.py 的如下部分:

```
DATABASES = {
    'default': {
        'ENGINE': 'sentry.db.postgres',
        'NAME': 'sentry',
        'USER': 'ubuntu',
        'PASSWORD': '123',
        'HOST': '',
        'PORT': '',
    }
}
```

```
EMAIL_HOST = 'smtp.qq.com' # 发送邮件配置, 本例使用QQ邮箱
EMAIL_HOST_PASSWORD = 'SOME_PASSWORD' # 这个密码是QQ邮箱需要用于登录第三方客户端的
    授权码, 而不是QQ密码。详情请参考http://service.mail.qq.com/cgi-bin/help?subtype=1&id=28&no=1001256
EMAIL_HOST_USER = 'xxx@qq.com'
EMAIL_PORT = 25
EMAIL_USE_TLS = True
```

```
SERVER_EMAIL = 'xxx@qq.com'
BROKER_URL = 'redis://localhost:6379' # 消息队列设置
```

初始化数据库:

```
> sentry upgrade
```

这个过程会提示创建新用户:

```
Would you like to create a user account now? [Y/n]: y
```

```
Email: xxx@qq.com
Password:
Repeat for confirmation:
Should this user be a superuser? [y/N]: y
```



也可以使用“sentry createuser”创建超级用户。

启动 Sentry

现在 Sentry 就可以启动了：

```
> sentry start
```

Sentry 的 Web 服务端口是 9000，访问 <http://127.0.0.1:9000>，使用刚才创建的 xxx@qq.com 就可以作为管理员登录了。

Sentry 使用 Celery 作为任务执行的 Worker 框架，Celery 在后面有专门的章节讲解，这里先启动它：

```
> sentry celery worker # 执行Sentry生成的任务
```

Sentry 会生成很多任务，比如发送邮件，合并事件。需要启动定时任务进程，来创建任务：

```
> sentry celery beat
```



如果产生的异常太多又不想保留，可以通过定时任务定期清理旧数据。

```
$ crontab -l
0 3 * * * sentry cleanup --days=90 # 清理三个月前的记录
```

Sentry 有非常好的数据迁移的设计，升级 Sentry 非常方便。每次使用 pip 更新 Sentry 包之后执行升级命令“sentry upgrade”即可。

创建团队和项目

登录后，默认存在一个叫作 Sentry 的团队（Team）。我们先创建一个 Team，单击 <http://localhost:5000/sentry/> 右上角的 New Team 按钮，输入 Subject，如图 7.1 所示。

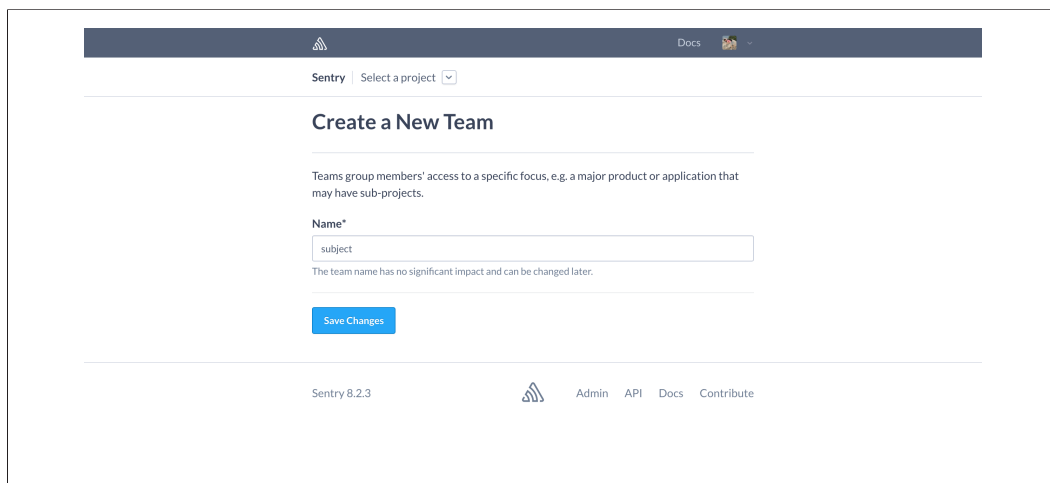


图 7.1 创建一个团队

单击 **Save Changes** 按钮进入创建项目页面（Project），输入项目名字 **r**，指定 Team 为 **subject**，如图 7.2 所示。

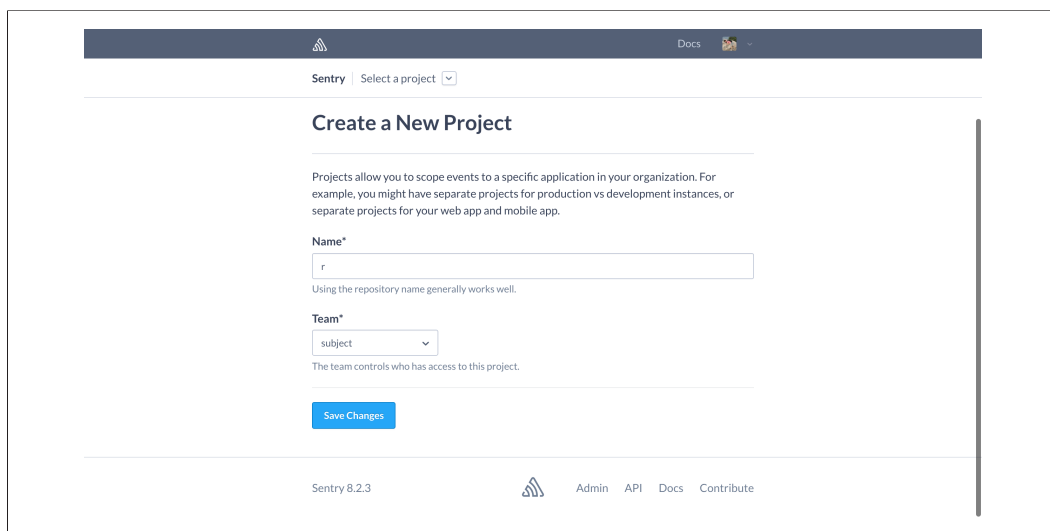


图 7.2 创建一个项目

再单击 **Save Changes** 按钮就完成了。

配置 SDK

安装 Sentry 的 Python 的 SDK:

```
> pip install raven
```

可以在 <http://localhost:5000/sentry/r/settings/keys/> 页面上找到 r 的 DSN (Data Source Name 的简称), 如图 7.3 所示。

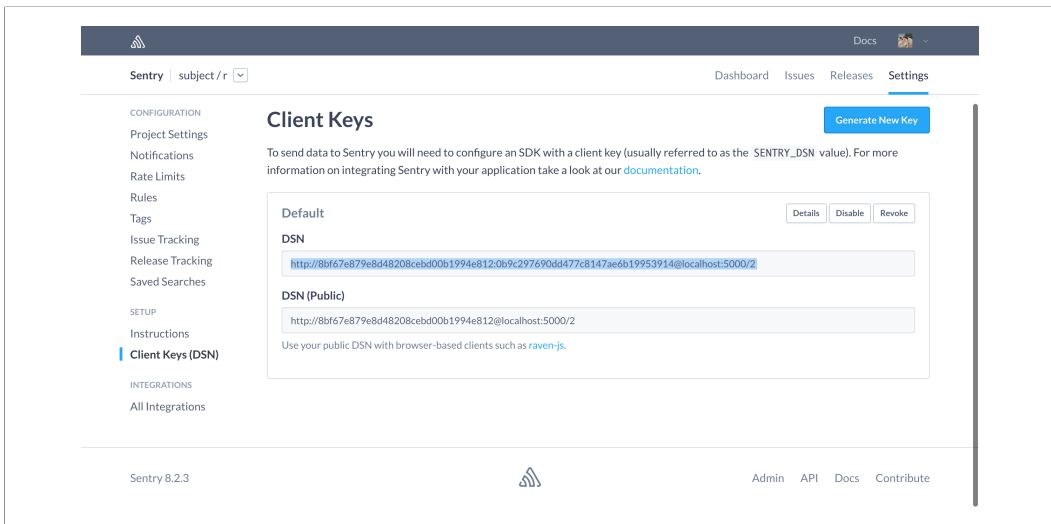


图 7.3 找到 r 的 DSN

创建一个简单的 Flask 应用 (app.py):

```
from flask import Flask
from raven.contrib.flask import Sentry

app = Flask(__name__)
sentry = Sentry(app, dsn='http://8bf67e879e8d48208cebd00b1994e812:0
b9c297690dd477c8147ae6b19953914@localhost:5000/2') # noqa
```

应用中包含三个视图。

1. `/error`: 对可预知的异常进行捕获, 它的优点是不影响用户访问。

```
@app.route('/error')
def error():
    try:
        1 / 0
    except ZeroDivisionError:
```

```
sentry.captureException()  
return 'error'
```

2. `/raise`: 未做异常处理的捕获, 一般不需要特别设置。只要出现不符合预期的异常, 理论上就会进入 Sentry:

```
@app.route('/raise')  
def auto_raise():  
    raise IndexError
```

3. `/log`: 捕获非异常信息, 可以把 Sentry 作为日志收集工具。

```
@app.route('/log')  
def log():  
    sentry.captureMessage('hello, world!')  
    return 'logging'
```

现在分别访问这三个页面。然后刷新一个 Sentry 的 `r` 项目主页 <http://localhost:5000/sentry/r>, 就可以看到 3 条记录, 见图 7.4。

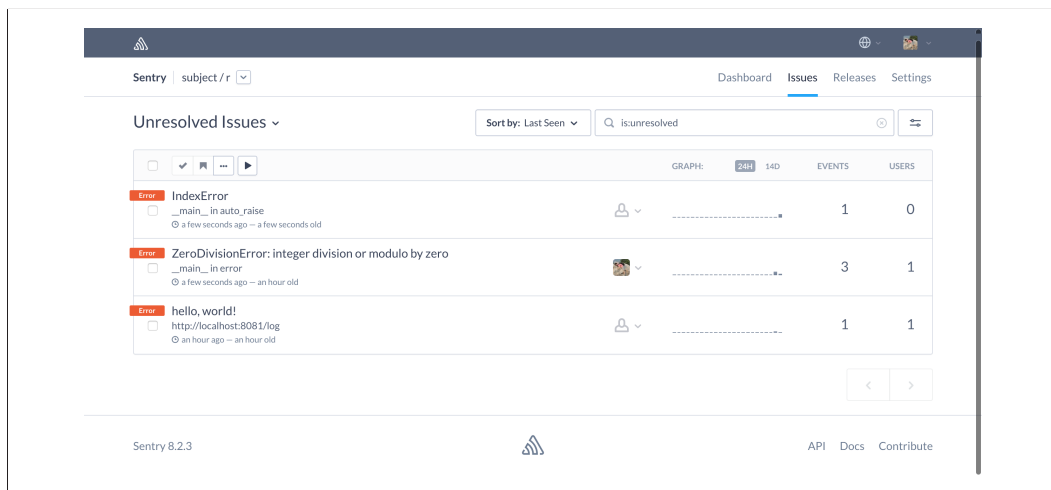


图 7.4 `r` 的项目主页上可以看到 3 条记录

看一下 `ZeroDivisionError` 这个异常的详情页, 如图 7.5 所示。

详情页直接定位错误的代码行数, 上下文以及相关的参数和变量。可以在 `Comments` 这个 Tab 下讨论问题, 对事件合并, 标记为解决/未解决等。

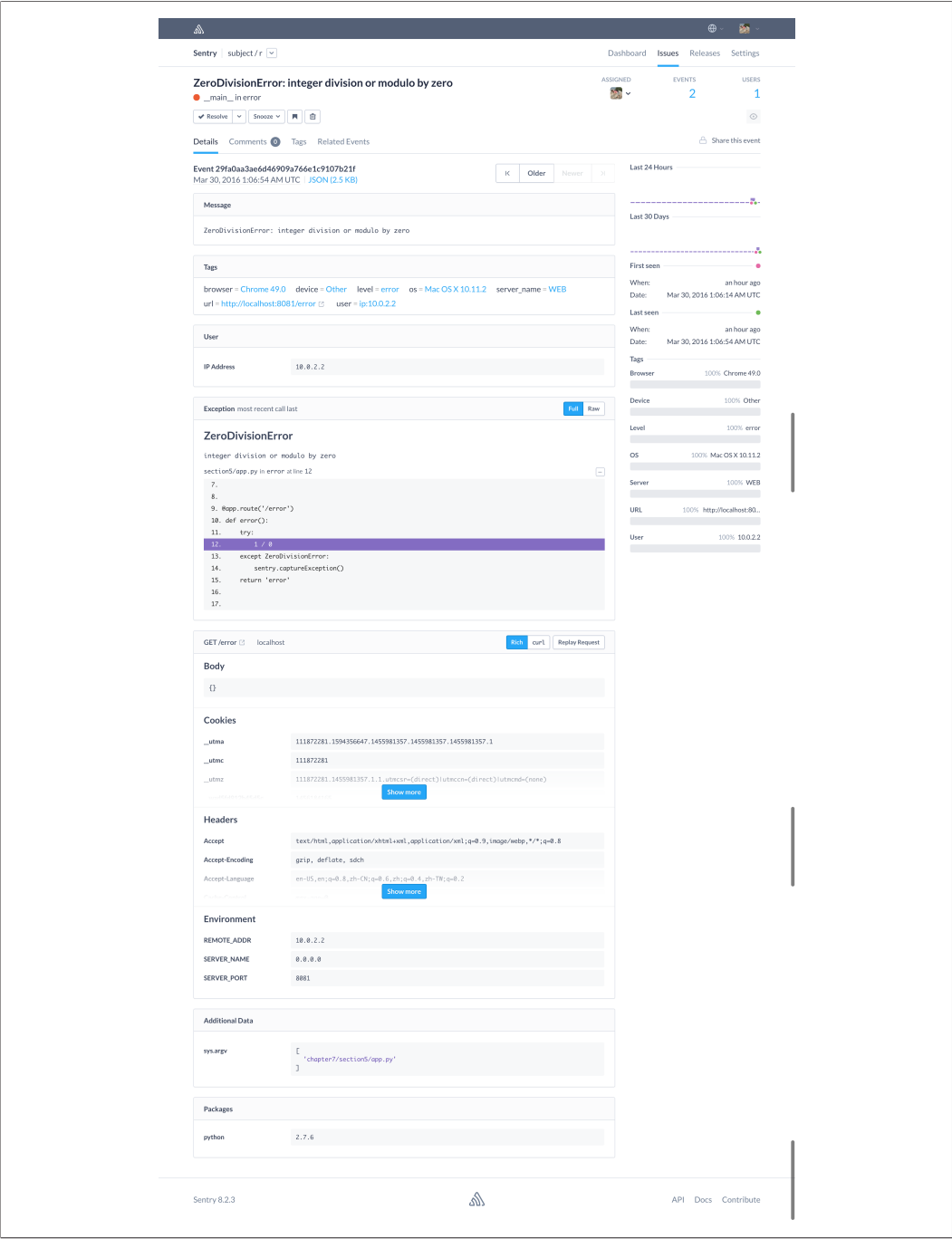


图 7.5 ZeroDivisionError 异常的详情页

收到的报警邮件如图 7.6 所示。

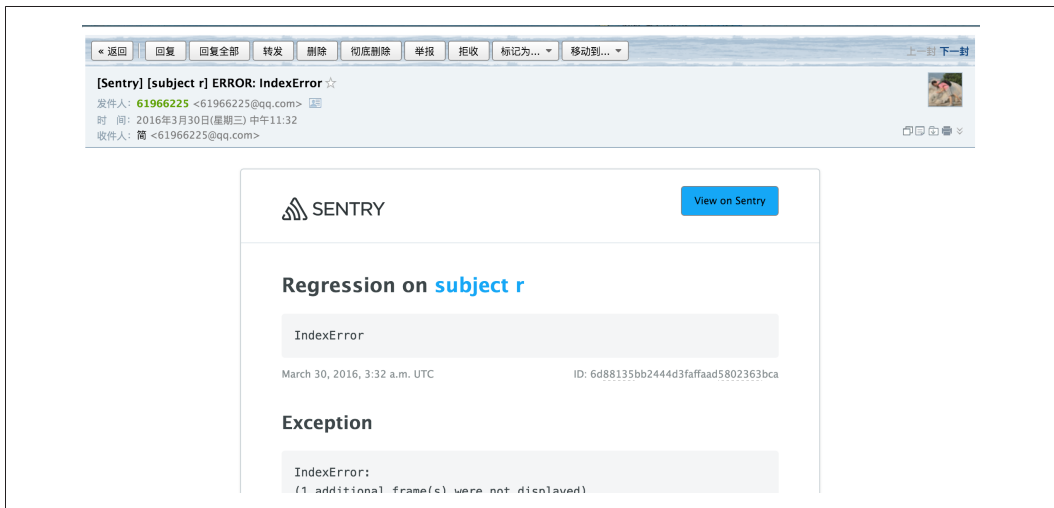


图 7.6 收到的报警邮件



Sentry 还可以和 Slack、Trello、GitHub、GitLab 等企业协作沟通工具相结合。

使用 StatsD、Graphite 等搭建 Web 监控

除了对应用程序的异常做监控，我们还需要对服务器的硬件、流量和应用相关服务运行情况做监控、搜集数据和绘制图表。通过搜集到的数据，我们可以发现当前系统的瓶颈，作为故障排除依据，决策何时扩展以及如何扩展等。

常见的运维监控工具及其主要应用场景如下。

- Zabbix: 诞生于 1998 年的企业级分布式监控系统，可以自动发现、自动注册服务器、实时绘图、存储历史数据。
- Nagios: 通过安装插件和编写监控脚本可以很容易地实现绝大部分的定制需求，它轻量灵活，且报警机制很强，但是在监控的服务器数量较多时会有性能瓶颈。Nagios 在绘制图表方面很弱，常使用 Cacti 补充。Cacti 是使用 PHP 语言开发的网络流量监测图形分析工具，用 RRDTool 存储和更新数据，当用户需要查看数据的时候用 RRDTool 生成图表呈现给用户。

- Icinga: 最初是 Nagios 的一个分支, 1.x 版本和 Nagios 功能基本一样, 只是添加了 Icinga-Web 的 REST API 等。
- Icinga2: 在 Icinga2 (2.x) 版本下重写了 Icinga, 它默认就支持 Graphite, 使用多线程来提高运行效率 (官网称 1 分钟百万级别地动态检查监控 6 万台主机没有卡顿), 使用全新的 Icinga-Web 2, 还支持可拖放的定制仪表盘。

对于服务器的硬件、流量和应用相关服务运行情况的监控和报警, 推荐使用 Icinga2。而绘制图表, 则选择更流行的 StatsD + Graphite + Grafana + Diamond 组合。

- Graphite: 一个基于 Django 的企业级监控工具, 能实时可视化和按时间序列存储数据。严格地说, Graphite 只是一个根据数据实时绘图的工具, 数据收集通常由第三方工具 (如 Ganglia、Collectd、StatsD、Diamond、Bucky 等) 或插件完成, 还可根据其协议选用别的数据源供其绘图。我们这里只使用它做数据库存储。
- StatsD: 它是一个 NodeJS 的 Daemon 程序, 最初由 Flickr 实现, Etsy 将其重构。StatsD 运行在每台服务器上, 服务器上的程序发送各种关键指标到本地 StatsD 进程, 最后 Graphite 根据 StatsD 聚合的数据画图。
- Diamond: 一个 Python 的指标收集守护程序, 它自带了 CPU、内存、网络、系统负载、磁盘等方面的指标收集器, 也非常容易地实现自定义的收集器。
- Grafana: 一个开源、功能强大、可定制性高的指标仪表板和图形编辑器工具。不直接使用 Graphite 展示图表, 是因为它的界面和功能过于简单、页面样式比较陈旧。使用 Grafana 来汇集 Graphite 的测量数据, 并以图形方式显示是个很好的选择。

Graphite 包含如下三个组件。

- Carbon: 基于 Twisted 的进程, 接收时间序列数据。Twisted 框架让它能够以很低的开销处理大量的客户端和流量。
- Whisper: 专门存储时间序列类型数据的小型数据库。
- Graphite-web: 一个基于 Django 的可以高度扩展的实时画图系统, 还提供查询数据的 API。

安装 Graphite 的步骤如下:

```
> sudo mkdir /opt
> sudo chown ubuntu:ubuntu /opt -R
> sudo apt-get install libcairo2-dev
> git clone git://git.cairographics.org/git/py2cairo
> ./autogen.sh && make && sudo make install
> pip install carbon --install-option="--prefix=/opt/graphite" --install-option="--install-lib=/opt/graphite/lib"
> pip install graphite-web --install-option="--prefix=/opt/graphite" --install-option
```

```
="--install-lib=/opt/graphite/webapp"
> pip install cairocffi whisper django==1.8.11 # 目前Graphite对1.9支持有限, 先使用1
.8的最高版本
```

配置 Graphite

```
> cd /opt/graphite/webapp/graphite
> cp local_settings.py.example local_settings.py
> cd /opt/graphite/conf/
> cp carbon.conf.example carbon.conf
> cp storage-schemas.conf.example storage-schemas.conf
> cp graphite.wsgi.example ../webapp/graphite/graphite_wsgi.py
```

其中 storage-schemas.conf 在中间添加一项 statsd:

```
> cat storage-schemas.conf
[carbon]
pattern = ^carbon\. # 匹配以carbon字符串开头的指标项名称
retentions = 60:90d # 数据点每60秒记录一次, 并保存90天

[statsd]
pattern = ^stats.* # 匹配以stats字符串开头的指标项名称
retentions = 10s:1d,1m:7d,10m:1y # 数据点每10秒记录一次, 并保存1天; 数据点每1分钟
    记录一次, 并保存7天; 数据点每10分钟记录一次, 并保存1年

[default_1min_for_1day]
pattern = .* # 捕捉与Carbon和Statsd无关的全部指标项
retentions = 60s:1d # 数据点每60秒记录一次, 并保存1天
```

默认 Graphite 使用 SQLite 存储数据库, 我们换成 MySQL, 使用之前的数据库 r。修改 local_settings.py 的 DATABASES 的设置:

```
DATABASES = {
    'default': {
        'NAME': 'r',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'web',
        'PASSWORD': 'web',
        'HOST': 'localhost',
        'PORT': ''
    }
}
```

生成 Graphite 需要的表结构:

```
> python manage.py syncdb
> python manage.py migrate
```

使用 createsuperuser 创建一个超级管理员, 未来使用它登录 Graphite-Web:

```
> python manage.py createsuperuser
```

Carbon 包含 3 个守护进程。

- carbon-cache.py: 接收指标项数据, 并用底层的 Whisper 库按照指定的时间间隔将这些值写入磁盘。
- carbon-relay.py: 用来复制和分片。
- carbon-aggregator.py: 运行于 carbon-cache.py 前面, 在 Whisper 中记录指标项之前, 将这些指标项缓存一段时间。通常用在不需要细粒度报告的场景下。

本节仅展示单机的简单模式, 因此只需要启动 carbon-cache.py 进程:

```
> cd /opt/graphite/
> ./bin/carbon-cache.py start
```

启动 Graphite-Web:

```
> cd /opt/graphite/webapp/graphite
> gunicorn graphite_wsgi:application -b 0.0.0.0:9000
```

在实际生产环境中, 应该使用之前介绍的 Nginx + Gunicorn 或者 Nginx + uWSGI 的组合。

使用 StatsD

```
> sudo apt-get install nodejs -y # 需要先安装Node.js
> cd /opt
> git clone git://github.com/etsy/statsd.git
> cd statsd
> cp exampleConfig.js Config.js
```

Config.js 改成:

```
{
  graphitePort: 2003,
  graphiteHost: "localhost",
  port: 8125,
  backends: [ "./backends/graphite" ]
}
```

启动 StatsD:

```
> nodejs stats.js Config.js
```

配置 Diamond

```
> pip install diamond --install-option="--prefix=/opt/diamond"
> cd /opt/diamond/etc/diamond
> cp diamond.conf.example diamond.conf
> mkdir /opt/diamond/run
> sudo mkdir /var/log/diamond
> sudo chown ubuntu:ubuntu /var/log/diamond -R
```

需要修改 diamond.conf 的如下配置:

```
[server]
handlers = diamond.handler.stats_d.StatsdHandler
pid_file = /opt/diamond/run/diamond.pid
collectors_path = /opt/diamond/share/diamond/collectors/
collectors_config_path = /opt/diamond/etc/diamond/collectors/
handlers_config_path = /opt/diamond/etc/diamond/handlers/
handlers_path = /opt/diamond/share/diamond/handlers/
```

发布指标项

指标项 (metric) 是一种随着时间不断变化的可度量的值。例如内存占用情况、每秒请求数、请求处理时间等。发给 Carbon 的数据包含如下三种信息:

- 指标项名称。
- 度量值。
- 时间序列, 通常是时间戳。

可以通过如下命令把指标直接发送到 Carbon 上:

```
> echo "carbon.agents.web.metricsReceived 28198 `date +%s`" | nc localhost 2003
```

而发送给 StatsD 的格式则有些变化:

```
> echo "sampling:1|c|@0.1" | nc -u -w0 127.0.0.1 8125
```

type_specification 用来聚合类型, 不需要添加时间戳字段。StatsD 提供了额外的聚合指标功能, 每次在设置的 flushInterval 的时间 (默认是 10 s) 超时后聚合, 然后发送给 Carbon 或者其他后端服务。

1. Counting: 使用 c 表示, 也就是一个 sum 操作。

```
> echo "counter:1|c" | nc -u -w0 127.0.0.1 8125
```

2. Sampling: 取样。下例表示取 1/10 的结果。

```
> echo "sampling:1|@0.1" | nc -u -w0 127.0.0.1 8125
```

3. Timing: 定时收集。下例表示收集 1 s 内的指标, 再取 1/10 的结果。

```
> echo "timing:1000|ms|@0.1" | nc -u -w0 127.0.0.1 8125
```

生成的指标包含 mean、sum、count、min、max 等几种类型, 后面还可以使用百分比 (默认是 90, 也就是 90%)。下面用代码展示它们的含义:

```
In : dataset = [9, 89, 44, 97, 80, 62, 63, 22, 81, 27] # 一个无序的数据集
In : sorted_dataset = sorted(dataset) # 先排序
In : percentile = 90
In : percentile_90 = sorted_dataset[:int(len(sorted_dataset) * percentile / 100)] # 获得最好的90%的指标
In : percentile_90
Out: [9, 22, 27, 44, 62, 63, 80, 81]
In : lower_90 = min(percentile_90) # 最小值 9。lower和lower_90其实是一个意思
In : mean_90 = mean(percentile_90) # 平均值 48.5
In : upper_90 = max(percentile_90) # 最大值 90
In : sum_90 = sum(percentile_90) # 总数 388
```

1. Gauges: 如果发送的指标没有更新, 就会一直发送这个值, 还可以通过相对的计算来更新而不是直接设置为其他值。

```
> echo "gauges:1000|g" | nc -u -w0 127.0.0.1 8125
> echo "gauges:-20|g" | nc -u -w0 127.0.0.1 8125 # 发送980
> echo "gauges:+10|g" | nc -u -w0 127.0.0.1 8125 # 发送990
```

2. Sets: 只记录去重的集合。已经存在的指标值不会被发送, 下例只会发送两个指标。

```
> echo "sets:20|s" | nc -u -w0 127.0.0.1 8125
> echo "sets:20|s" | nc -u -w0 127.0.0.1 8125
> echo "sets:19|s" | nc -u -w0 127.0.0.1 8125
```

3. 多指标: 我们可以把上述 5 种指标组合, 使用\n 分隔:

```
> echo "counter:1|c\ntiming:320|ms\ngauges:-20|g\nuniques:111|s" | nc -u -w0 127.0.0.1 8125
```

使用 Grafana

登录 graphite-Web 就可以看到我们发送的这些指标了, 如图 7.7 所示。

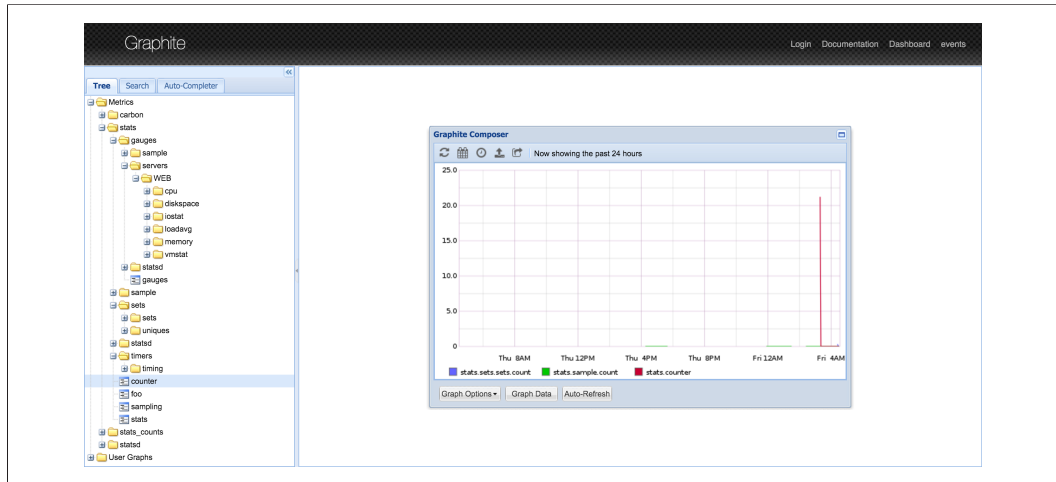


图 7.7 发送的指标

其中 Diamond 产生的指标都放在 Metrics->stats->gauges->servers 下。页面比较粗糙，现在使用 Grafana 来代替：

```
> sudo apt-get install -y adduser libfontconfig
> cd /tmp
> wget https://grafanarel.s3.amazonaws.com/builds/grafana_2.6.0_amd64.deb # 使用apt安
    装很慢，换成deb包的方式安装
> sudo dpkg -i grafana_2.6.0_amd64.deb
```

修改/etc/grafana/grafana.ini 中如下的设置：

```
[database]
# Either "mysql", "postgres" or "sqlite3", it's your choice
type = mysql
host = 127.0.0.1:3306
name = r
user = web
password = web
[server]
http_port = 5000
[security]
admin_user = dongwm
admin_password = 123
```

启动服务：

```
> sudo /etc/init.d/grafana-server start
```

访问 `http://localhost:5000`, 用户名 `dongwm`, 密码为 `123`。先创建数据源。访问 `http://localhost:5000/datasources/new`。填好类似的如图 7.8 所示表单。

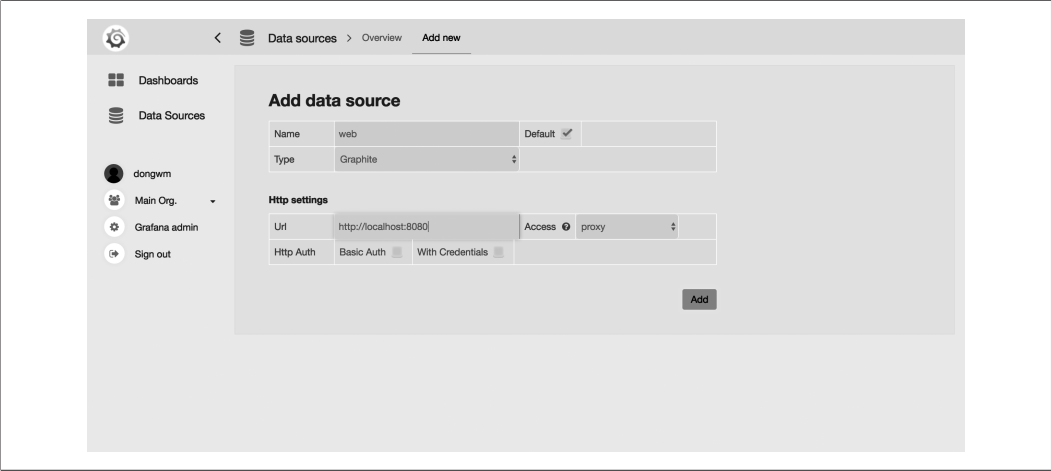


图 7.8 表单

单击右下角的 “Add”，提示添加成功。
然后创建绘图面板。打开首页 `http://localhost:5000`，打开如图 7.9 所示的 Home 下拉框。

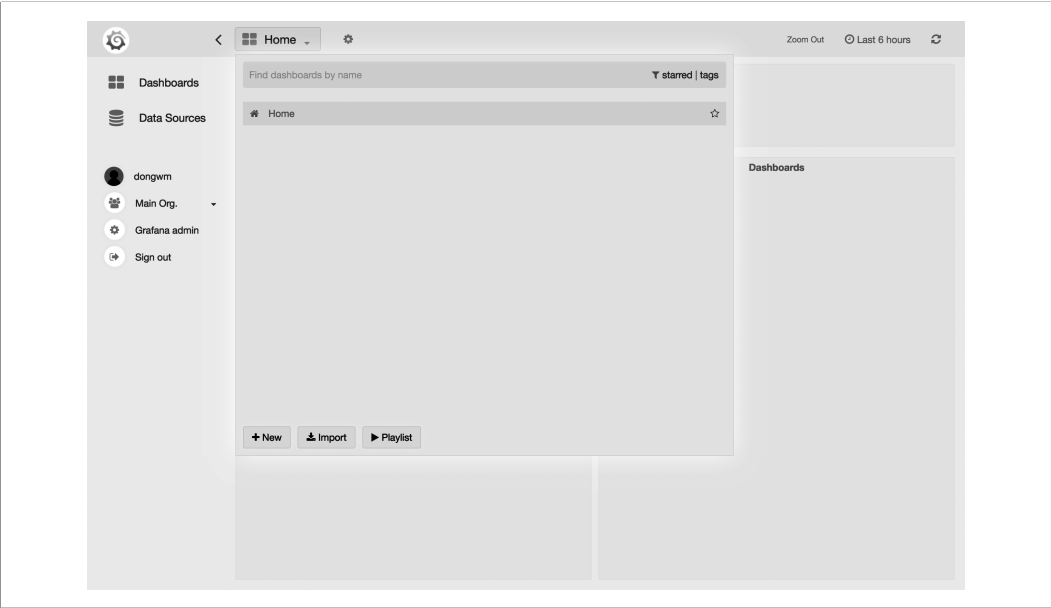


图 7.9 打开 Home 下拉框

单击最下面的“New”按钮。也可看到页面中间靠上的位置有一个绿色的小长块，打开它，选择 Add Panel->Graph，如图 7.10 所示。

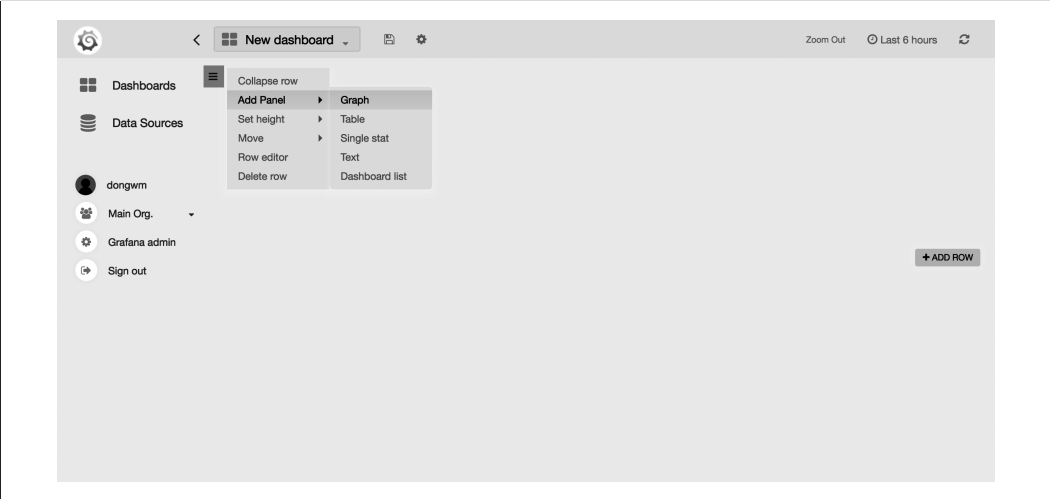


图 7.10 选择 Add Panel->Graph

然后就进入了面板创建页。一个面板可以添加多个指标。我们添加 Diamond 创建的几个指标，如图 7.11 所示。

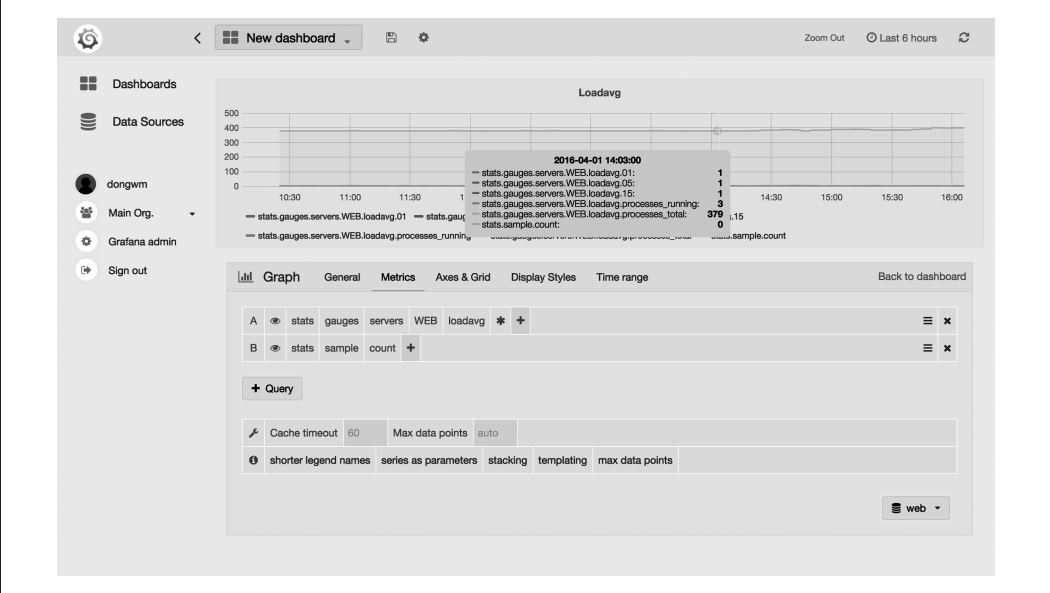


图 7.11 添加 Diamond 创建的几个指标

Query A 使用星号 “*” 匹配的全部类型，Query B 是为了演示而添加的。可以单击 + Query 按钮添加更多的 Query。



面板的标题等内容可以通过编辑原始 JSON 元数据来设置。

这样一个面板就做好了，如图 7.12 所示。

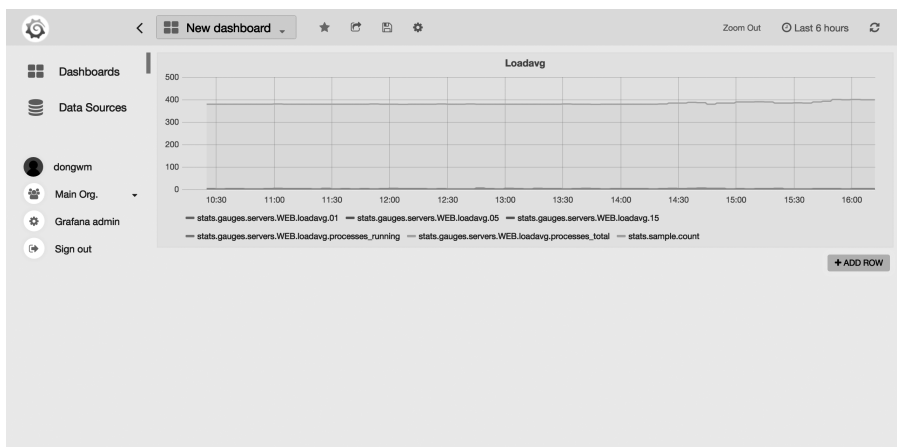


图 7.12 做好的面板

可以通过单击 “ADD ROW” 添加更多的面板。

使用 Kenshin

Kenshin (<https://github.com/douban/Kenshin>) 是豆瓣开源的时间序列数据库，它可用来替代 Whisper，IOPS (Input/Output Operations Per Second，即每秒进行读写〈I/O〉操作的次数) 提高了 40 倍。

还可以使用 carbon-c-relay 替代 carbon-relay，让 CPU 使用率降为原来的 10%。

第 8 章

测试和持续集成

测试指的是通过编写独立于业务代码的测试代码来验证程序中是否有错误。为什么需要测试呢？笔者认为有以下几点原因：

- 测试可以保证代码在预想到的情况下正常工作。
- 确保对代码的改动不会破坏现有的功能。
- 良好的测试要求业务代码模块化，代码耦合度低，这在一定程度上保证了业务代码的质量。

在编写业务代码的同时把测试代码补齐是一种非常好的习惯。这样既能保证测试覆盖了目前的业务场景，也能帮助其他维护者通过测试代码了解业务代码。

本章将展开测试和持续集成主题，主要包含如下内容：

- 介绍 Python 内置测试模块 `unittest` 和 `doctest`。
- 介绍第三方测试工具 `pytest` 和 `mock`。
- 深入持续集成，并通过 Buildbot 实际地对一个 GitHub 项目进行集成。

使用 `unittest` 和 `doctest` 做测试

`unittest`

`unittest` 是 Python 标准库里用于单元测试的模块。单元测试用来对最小可测试单元进行正确性检验。产品开发需要快速迭代功能，排期很紧，而写测试看起来是一件很麻烦的事。单元测

试的价值就在于维护现有功能时，尤其是不熟悉现有功能的新人，可以通过单元测试确认对代码的修改是否引入了新的错误或导致旧代码产生错误，帮助我们在上线之前就发现问题。

Python 标准库和优秀的 Python 项目都附带测试代码，如果一个项目不包含测试代码，很难说服用户它是可信赖的，继而在生产环境中去使用它。笔者在阅读项目源代码的时候，如果遇到文档匮乏或者表述不清、代码晦涩难懂，也会通过查看测试用例来理解。

通过测试 collections 模块中的 Counter 类，先来了解 unittest 的用法（ut_case.py）：

```
import unittest

from collections import Counter

class TestCounter(unittest.TestCase):
    def setUp(self):
        self.c = Counter('abcdaba')
        print 'setUp starting ...'

    def test_basics(self):
        c = self.c
        self.assertEqual(c, Counter(a=3, b=2, c=1, d=1))
        self.assertIsInstance(c, dict)
        self.assertEqual(len(c), 4)
        self.assertIn('a', c)
        self.assertNotIn('f', c)
        self.assertRaises(TypeError, hash, c)

    def test_update(self):
        c = self.c
        c.update(f=1)
        self.assertEqual(c, Counter(a=3, b=2, c=1, d=1, f=1))
        c.update(a=10) # a是累加的
        self.assertEqual(c, Counter(a=13, b=2, c=1, d=1, f=1))

    def tearDown(self):
        print 'tearDown starting ...'

if __name__ == '__main__':
    unittest.main()
```

setUp 方法列出了测试前的准备工作，常用来做一些初始化，非必需方法。tearDown 方法列出了测试完成后的收尾工作，用来销毁测试过程中产生的影响，非必需方法，但是应该合理使用。

TestCase，顾名思义表示测试用例，一个测试用例可以包含多个测试方法，每个方法都要以 test_ 开头。测试方法中用到的 self.assertXXX 的方法是断言语句，单元测试都是使用这样的断言语句判断测试是否通过的：如果断言为 False，会抛出 AssertionError 异常，测试框架就会认为此测试用例测试失败。

运行一下：

```
> python chapter8/section1/ut_case.py
setUp starting ...
tearDown starting...
.setUp starting ...
tearDown starting ...
.
-----
Ran 2 tests in 0.004s
```

OK

可以看到每次执行 test_ 开头的方法时都会执行 setUp 和 tearDown。

如果测试失败，会列出出错的详细行数、实际结果和预期结果等帮助我们定位问题。现在修改第 16 行，让它失败：

```
> python chapter8/section1/ut_case.py
...
=====
FAIL: test_basics (__main__.TestCounter)
-----
Traceback (most recent call last):
  File "chapter8/section1/ut_case.py", line 16, in test_basics
    self.assertEqual(len(c), 1)
AssertionError: 4 != 1
-----
Ran 2 tests in 0.002s
```

FAILED (failures=1)

另一个常见的测试用法是使用 unittest.TestSuite（测试套件），它将一组测试用例作为一个测试对象（ut_suite.py）：

```
import unittest

from collections import Counter

class TestCounter(unittest.TestCase):
```

```
def setUp(self):
    self.c = Counter('abcdaba')
    print 'setUp starting ...'

def runTest(self): # 需要重载这个方法
    c = self.c
    self.assertEqual(c, Counter(a=3, b=2, c=1, d=1))

def tearDown(self):
    print 'tearDown starting ...'

if __name__ == '__main__':
    suite = unittest.TestSuite()
    suite.addTest(Counter()) # 可以用addTest添加更多的TestCase实例
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

还可以更细粒度地添加测试用例中的一部分方法 (ut_suite_with_case.py):

```
import unittest
from ut_case import Counter

if __name__ == '__main__':
    suite = unittest.TestSuite()
    suite.addTest(Counter('test_basics'))
    suite.addTest(Counter('test_update'))
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

笔者对单元测试的理解如下:

- 软件质量不是测试出来的, 而是设计和维护出来的。
- 并不是所有的模块都需要添加单元测试, 单元测试应该用来测试那些可能会出错的地方。与其追求代码覆盖率, 不如将重点关注在确保写出更有质量的业务代码和更有意义的测试上。
- 保持测试的独立性。测试用例之间最好不要有相互依赖, 也不能依赖执行的先后次序。

doctest

doctest 模块是一种非常直观的表述型测试方法, 它通常查找代码文件里文档字符串中的交互式会话部分, 执行那些会话以验证代码工作是否正常 (doc_test.py):

```
def import_object(name):
    """Imports an object by name.
    >>> import os.path
    >>> import_object('os.path') is os.path
    True
    >>> import_object('os.missing_module')
    Traceback (most recent call last):
    ...
    ImportError: No module named missing_module
    """

    parts = name.split('.')
    obj = __import__('.', '.join(parts[:-1]), None, None, [parts[-1]], 0)
    try:
        return getattr(obj, parts[-1])
    except AttributeError:
        raise ImportError('No module named {}'.format(parts[-1]))

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

也可以使用-m 的方式执行：

```
python -m doctest chapter8/section1/doc_test.py
```

没有输出就表示测试通过。



给功能函数添加 doctest 是一个好习惯。

使用 py.test 和 mock

Python 标准库提供的测试模块功能相对单一，所以在项目中通常会额外使用第三方的测试工具。

py.test

py.test (<https://pytest.org>) 也叫作 pytest，除了比 Python 标准的单元测试模块 unittest 更简洁和高效外，还有如下特点：

- 容易上手，入门简单，文档中有很多实例可以参考。
- 可以自动发现需要测试的模块和函数。
- 支持运行由 nose、unittest 等模块编写的测试用例。
- 有很多第三方插件，并且可以方便地自定义插件。
- 很容易与持续集成工具结合。
- 可以细粒度地控制要测试的测试用例。

我们先安装它：

```
> pip install pytest
```

下面演示 py.test 常用的测试方法（test_pytest.py）：

```
import pytest
```

```
@pytest.fixture # 创建测试环境，可以用来做setUp和tearDown的工作
```

```
def setup_math():  
    import math  
    return math
```

```
@pytest.fixture(scope='function')
```

```
def setup_function(request):  
    def teardown_function():  
        print("teardown_function called.")  
    request.addfinalizer(teardown_function) # 这个内嵌的函数做tearDown工作  
    print('setup_function called.')
```

```
def test_func(setup_function):  
    print('Test_Func called.')
```

```
def test_setup_math(setup_math):  
    # py.test不需要使用self.assertXXX这样的方法，直接使用Python内置的断言语句  
    # assert即可  
    assert setup_math.pow(2, 3) == 8.0
```

```
class TestClass(object):  
    def test_in(self):  
        assert 'h' in 'hello'  
  
    def test_two(self, setup_math):
```



```

    assert setup_math.ceil(10) == 10.0

def raise_exit():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit): # 用来测试抛出的异常
        raise_exit()

@pytest.mark.parametrize('test_input,expected', [
    ('1+3', 4),
    ('2*4', 8),
    ('1 == 2', False),
]) # parametrize可以用装饰器的方式集成多组测试样例
def test_eval(test_input, expected):
    assert eval(test_input) == expected

```

unittest 必须把测试放在 TestCase 类中，pytest 只要求函数或者类以 test 开头即可。

运行一下：

```

> py.test chapter8/section2/test_pytest.py
===== test session starts =====
platform linux2 -- Python 2.7.11 -- py-1.4.31 -- pytest-2.6.4
plugins: django
collected 8 items

chapter8/section2/test_pytest.py .....

===== 8 passed in 4.04 seconds =====

```

测试通过了。我们让其中一个测试失败：

```

> py.test
===== test session starts =====
platform linux2 -- Python 2.7.6 -- py-1.4.31 -- pytest-2.6.4
plugins: django
collected 8 items

test_pytest.py ...F....

===== FAILURES =====
_____TestClass.test_two _____

```

```

self = <test_pytest.TestClass object at 0x7fe8de0ba210>
setup_math = <module 'math' (built-in)>

    def test_two(self, setup_math):
>     assert setup_math.ceil(10) == 11.0
E     assert 10.0 == 11.0
E       + where 10.0 = <built-in function ceil>(10)
E       +   where <built-in function ceil> = <module 'math' (built-in)>.ceil

test_pytest.py:34: AssertionError
===== 1 failed, 7 passed in 0.12 seconds =====

```

py.test 帮助我们定位到测试失败的位置，并告诉我们预期值和实际值。py.test 的命令行功能非常丰富：

```

# 和使用py.test的作用一样
> python -m pytest chapter8/section2/test_pytest.py
# 验证整个目录
> py.test chapter8/section2
# 只验证文件中单个测试用例，这在实际工作中非常方便，否则可能需要运行一段时间才能
  轮到到有问题的测试用例，极为浪费时间。使用这样的方式就可以有针对性地验证有
  问题的测试用例
> py.test chapter8/section2/test_pytest.py::test_mytest
# 只验证测试类中的单个方法
> py.test chapter8/section2/test_pytest.py::TestClass::test_in

```

插件

py.test 有丰富的第三方插件，如下 3 个插件很有用。

1. pytest-random：让测试用例的测试顺序变成随机的。当有很多测试用例时，这个插件不会让测试只卡在一个异常上，有助于发现其他异常。

```

> pip install pytest-random # 插件的名字都以`pytest-`开头
> py.test --random

```

2. pytest-xdist：让 py.test 支持分布式的测试。

```

> pip install pytest-xdist
# 把测试分发到多个CPU上执行，如果测试花费的时间太久以及需要大量I/O操作，
  使用多个CPU能减少很多时间
> py.test -n 3 chapter8/section2/test_pytest.py
# 使用子进程进行并发测试
> py.test -d --tx 3\*popen//python=python2.7 chapter8/section2/test_pytest.py

```

3. `pytest-instafail`: 一旦测试出现出错信息就立即返回, 不需要等全部测试结束后才显示。

```
> pip install pytest-instafail
> py.test --instafail
```



你可能听过或者在用 Nose, 它最近几年一直处于维护状态而且可能停止, 应该使用 `py.test` 替换掉它。

mock

Mock 测试是在测试过程中对可能不稳定、有副作用、不容易构造或者不容易获取的对象, 用一个虚拟的对象来创建以便完成测试的方法。在 Python 中这种测试是通过第三方的 `mock` 库完成的, `mock` 在 Python 3.3 的时候被引入到 Python 标准库中, 改名为 `unittest.mock`。之前的 Python 版本都需要安装它:

```
> pip install mock
```

假设现在一个单元测试依赖外部的 API 返回的值。举个例子 (`client.py`):

```
import requests

def api_request(url):
    r = requests.get(url)
    return r.json()

def get_review_author(url):
    rs = api_request(url)
    return rs['review']['author']
```

如果测试时每次都真正去请求这个接口, 就会有两个问题:

- 测试环境可能和线上环境不同, 需要搭建本地的 API 服务, 尤其是需要本地环境能返回线上环境实际的全部结果, 增加复杂度且效率低下。
- 测试结果严重依赖外部 API 服务的稳定性。

使用 `mock` 的解决方案如下 (`test_mock.py`):

```
import unittest

import mock
```

```
import client
```

```
class TestClient(unittest.TestCase):
    def setUp(self):
        self.result = {'review': {'author': 'dongwm'}}

    def test_request(self):
        api_result = mock.Mock(return_value=self.result)
        client.api_request = api_result
        self.assertEqual(client.get_review_author(
            'http://api.dongwm.com/review/123'), 'dongwm')
```

这个测试并没有实际地请求 API 就达到了测试的目的。还可以通过 `mock.patch` 来创建：

```
def test_request(self):
    api_result = mock.Mock(return_value=self.result)
    with mock.patch('client.api_request', api_result):
        self.assertEqual(client.get_review_author(
            'http://api.dongwm.com/review/123'), 'dongwm')
```

使用 `patch` 的目的是为了控制 `mock` 的范围，使其在 `patch` 上下文之外不受影响，这样使用更灵活。`mock.patch` 除了用作上下文管理器，还可以作为装饰器加在测试方法上：

```
@mock.patch('client.api_request')
def test_request(self, api_request): # 每个patch装饰器会作为一个参数传进来
    api_request.return_value = self.result
    self.assertEqual(client.get_review_author(
        'http://review.dongwm.com/dongwm'), 'dongwm')
```

当初始化 `mock.Mock` 类之后，可以对任何属性及其子属性进行预设。另一个常见的模拟场景是数据库操作。下面模拟 MySQL 的查询语句：

```
In : mock = Mock()
In : cursor = mock.connection.cursor.return_value
In : cursor.execute.return_value = (1, 'xiaoming')
In : mock.connection.cursor().execute('SELECT * from users limit 1')
Out: (1, 'xiaoming')
In : mock.connection.cursor().execute('show tables') # 无论参数是什么，都会返回这个值
Out: (1, 'xiaoming')
```

当被模拟的函数（方法）有参数且参数对返回值有影响，或者抛出异常时，使用 `side_effect`：

```
def test_side_effect():
    mock = Mock()
```

```
def effect(*args, **kwargs):
    raise IndexError

mock.side_effect = effect
with pytest.raises(IndexError):
    mock(1, 2, a=3)

side_effect = lambda value, length=1: value * length
mock.side_effect = side_effect
assert mock(1) == 1
assert mock('*', 2) == '**'
```

持续集成

持续集成（Continuous Integration，CI）是一种软件开发实践。持续集成的目的简单而明确，就是让产品可以快速迭代，同时还能保持高质量。当有人向代码库的主分支提交代码时，持续集成服务器会尝试去构建整个产品，包括单元测试、Web 测试、代码质量分析等等。如果构建通过，说明改动的代码被成功地集成了，否则就说明本次构建有问题需要修复，在修复完成之前这些代码是不能被合并到主分支的。

持续表示会不断获取反馈，响应反馈。对于 Python Web 开发来说，集成主要是指测试和代码审查。不断地集成和修正集成的结果，直到达到集成要求。这种方式有如下优点：

- 更快地发现潜在问题，更容易定位错误。
- 实现测试自动化。
- 让整个团队高效准确地工作，可以有效避免由不同人开发的单个小模块可以单独工作，但集成为一个大系统则失败的问题。

CI 服务器根据源代码的变更触发构建，监控测试结果。目前常用的 CI 服务器有 5 种。

1. Jenkins：一个用 Java 编写的开源持续集成工具，它有丰富的插件和完善的 API，可以自由扩展。豆瓣的持续集成系统也基于 Jenkins。虽然它很受欢迎，但是 Python 工程师很难对其进行二次开发，页面和功能也有些陈旧。
2. Travis CI：一个针对 GitHub 的云服务平台，只对开源项目提供免费的 CI 服务。
3. GitLab CI：为 GitLab 提供持续集成服务的开源服务器。
4. Buildbot：用 Python 编写的一个开源 CI 系统。Google Chromium、Python、MongoDB、Mercurial、WebKit、Zope 等项目都使用它来作为持续集成工具。
5. Strider：一个使用 Node.js 开发的、开源持续集成和发布服务器。目前它已经支持 GitHub、Bitbucket、Gitlab 等平台，支持 Python、Ruby、Node.js 和其他自定义的应用。

本节将使用 Buildbot 实现 GitHub 开源项目的持续集成。首先在 GitHub 上创建一个用来测试的项目 <https://github.com/dongweiming/tola>。项目的结构如下：

```
├─ setup.cfg
├─ setup.py
├─ requirements.txt
├─ tests
├─ test_pytest.py
```

其中 test_pytest.py 使用 8.3 节的测试代码。setup.cfg 是 INI 格式的配置文件，可以把支持的命令的默认设置放入这个配置文件中，而不需要在命令行指定：

```
[pytest]
norecursedirs=venv # virtualenv创建的环境目录不需要验证
```

使用 “python setup.py test” 是常见的测试方式，需要创建 setup.py 文件：

```
from setuptools import find_packages, setup
from setuptools.command.test import test as TestCommand

class PyTest(TestCommand):

    def run_tests(self):
        import pytest
        errno = pytest.main([])
        exit(errno)

setup(
    name='Aiglos',
    ...
    packages=find_packages(
        exclude=["*.tests", "*.tests.*", "tests.*", "tests"]), # 最后安装的时候需要
        排除测试相关的文件
    zip_safe=True, # 出于性能考虑，通常可以把包打包成zip文件。如果包内没有数据文
        件、C扩展等就可以选择压缩
    test_suite='tests', # 指定测试套件的目录
    tests_require=['pytest'], # 测试需要安装的依赖，不需要使用install_requires
    cmdclass={'test': PyTest}, # 自定义了命令类
    ...
)
```

Buildbot 的环境数据都存在数据库中，本节将使用 MySQL 存储这些数据。Buildbot 的稳定版本是 0.8.12，但是此版本还不兼容 MySQL 5.7，需要使用更新的 0.9 版本：

```
> pip install buildbot==0.9.0b9 buildbot-www==0.9.0b9 buildbot-worker==0.9.0b9 buildbot
    -waterfall-view==0.9.0b9 buildbot-console-view==0.9.0b9
```

Buildbot 0.9 做了比较大的改动，现在 buildbot-slave 更名为 buildbot-worker，并把集成的 Web 应用独立成了 buildbot-www。buildbot-waterfall-view 和 buildbot-console-view 是额外的组件，分别用瀑布和终端视图的方式展示集成结果。

Master 是主控端，首先配置 Master：

```
> buildbot create-master -r /srv/build_bot
> cp /srv/build_bot/master.cfg.sample /srv/build_bot/master.cfg
```

更新 master.cfg 的如下配置：

```
# 修改项目相关信息，其中title和buildbotURL的设置会在邮件中展示出来
c['title'] = 'Tola'
c['titleURL'] = "https://github.com/dongweiming/tola"
c['buildbotURL'] = "http://localhost:9000/"

# 设置数据库的地址，其中参数sql_mode=MYSQL40是为了兼容MySQL 5.x
c['db'] = {
    'db_url': "mysql://web:web@localhost:3306/r?sql_mode=MYSQL40",
}

# 执行集成任务的worker设置
c['workers'] = [worker.Worker('slave1', '456')]

# 构建步骤。构建分两步：先克隆代码，然后执行python setup.py test
factory = util.BuildFactory()
factory.addStep(
    steps.Git(repourl='git://github.com/dongweiming/tola.git', mode='incremental'))
factory.addStep(steps.ShellCommand(command=["python", "setup.py", "test"]))
# 设置触发构建的条件，GitPoller会监控这个版本库，如果版本库有变化就会触发一次新的构建
c['change_source'] = []
c['change_source'].append(changes.GitPoller(
    'git://github.com/dongweiming/tola.git',
    workdir='gitpoller-workdir', branch='master',
    pollinterval=10))
# pollinterval表示检查间隔，默认是300s，时效性不够，这里改成了10s

# 构建设置，使用worker名字为slave1来执行构建，构建名字是runtests
c['builders'] = []
c['builders'].append(
    util.BuilderConfig(name="runtests",
                       workernames=["slave1"],
                       factory=factory))

# Buildbot的Web页面设置：使用端口9000，登录用户为dongwm，密码为123
c['www'] = {
```

```
'port': 9000,
'plugins': dict(waterfall_view={}, console_view={}),
'auth': util.UserPasswordAuth({'dongwm': '123'})
}

c['services'] = []

from buildbot.plugins import reporters

# 添加邮件监控设置, 接收邮件的是xxx@gmail.com和对项目感兴趣的作者
mn = reporters.MailNotifier(fromaddr="buildbot@mailgun.org",
                           relayhost="smtp.mailgun.org",
                           extraRecipients=['xxx@gmail.com'],
                           smtpUser="postmaster@xxx.mailgun.org",
                           smtpPassword="smtpPassword")

c['services'].append(mn)
```

其中发送监控邮件的 MailNotifier 类的 extraRecipients 通常是一个邮件组, 这样相关工程师就都可以收到邮件了。

启动 Master:

```
> buildbot upgrade-master /srv/build_bot # 更新配置都需要执行这一步
> buildbot start /srv/build_bot
```

接着创建和配置执行任务的 Worker:

```
> buildbot-worker create-worker /srv/build_bot_worker localhost:9989 slave1 456
```

Worker 不一定与 Master 在同一个服务器上。启动 Worker:

```
> buildbot-worker start /srv/build_bot_worker
```

现在 push 代码到 tola 项目就会触发自动构建了。访问 <http://localhost:9000>, 然后使用上面设置的 dongwm/123 登录。看一下最近的构建记录 (单击 “Recent Builds”), 如图 8.1 所示。

构建成功的效果如图 8.2 所示。

构建失败的效果如图 8.3 所示。

通过构建的 ID 可以看到构建过程, 比如看 ID 为 1 的 Shell 记录 (Shell 是配置的 factory.addStep 的第二步), 地址就是 <http://localhost:9000/#/builders/1/builds/4/steps/1/logs/stdio>。

我们也会收到包含构建结果的邮件, 如图 8.4 所示。

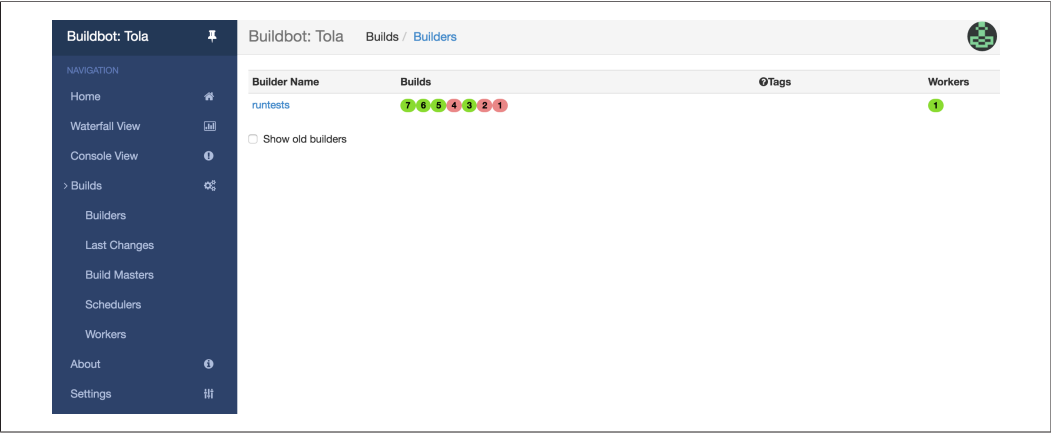


图 8.1 最近的构建记录

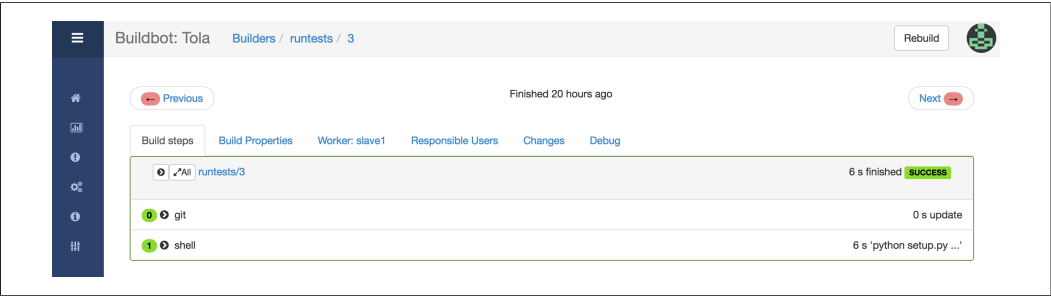


图 8.2 构建成功

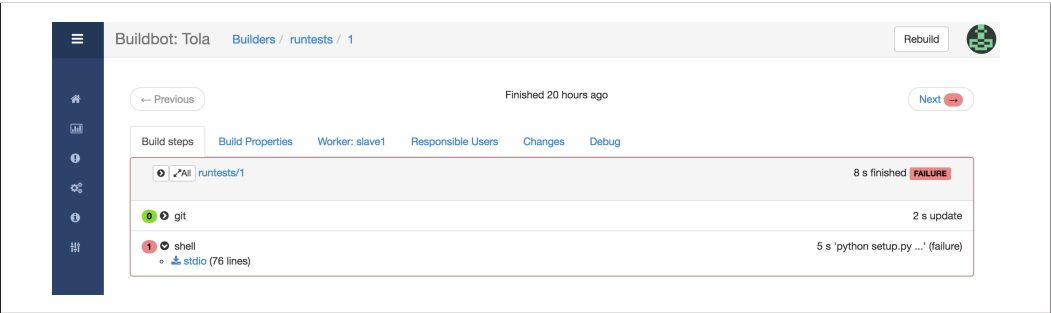


图 8.3 构建失败

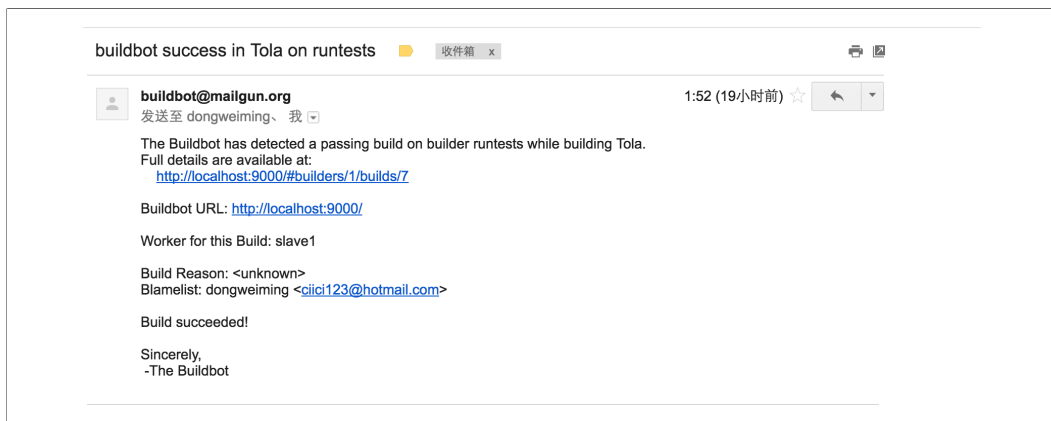


图 8.4 包含构建结果的邮件

使用 Tox 集成

Tox 是一个通用的 Virtualenv 管理和测试命令行工具，它的目标是提供最先进的自动化打包、测试和发布功能。Tox 只能在终端使用命令行的方式做测试，多用于本地环境的测试或者集成进 CI 服务器。它最大的特性是支持一次性验证多个版本的 Python 解释器下的包的集成情况，在开源项目中使用比较广泛。

我们先安装它：

```
> pip install tox
```

Tox 用到的目录结构：

```
> cd ~/web_develop/chapter8/section3
> tree
|— requirements.txt  # 依赖文件
|— tests
|   |— test_pytest.py  # 使用本章8.1节中Py.test小节的测试用例
|— tox.ini  # Tox的配置文件
```

Tox 配置如下：

```
[tox]
envlist = py{2.7,3.5}  # 是py27,py35的简写，也就是默认创建Python 2.7和Python 3.5这两个虚拟环境
skipdist = True  # Tox默认会使用sdist构建包，对于测试来说没有必要，而且构建还会要求存在README、setup.py等文件，并且保证setup.py的格式符合要求等，所以本例跳过了这一步
```

```
[testenv] # 默认的集成方案
deps = pytest # 集成需要的依赖
commands = {posargs:py.test} # 可以把调用的命令的参数通过posargs传给Tox来使用

[testenv:flake8] # 非默认的集成方案，需要使用tox -e flake8才能调用
basepython=python2.7 # 使用Python 2.7这个解释器完成工作
deps =
    pytest # 依赖可以多行
    -rrequirements.txt # 文件中有相关的其他依赖：flake8
commands =
    {posargs:py.test} # 相当于执行两步：第一步执行py.test，第二步验证代码规范问题
    flake8 .
```

可以使用如下命令集成：

- `tox` # 默认方案，会创建py27和py35这两个虚拟环境，目录默认在当前目录的`.tox`目录下，使用的命令就是`pytest`
- `tox -e flake8` # 使用`testenv:flake8`这个块中的集成方案
- `tox -e py2.7` # 只创建py27的集成方案，它和`tox -e py27`同义，但是由于之前执行`tox`的时候创建过，所以更节省时间
- `tox -- py.test -k test_eval` # 只测试了`test_eval`相关的三个测试，这就是`{posargs:py.test}`的作用
- `tox -e py27 -- python` # 进入Python 2.7的Python交互模式
- `tox -e py27 -- python -m SimpleHTTPServer` # 可以支持多个参数

第 9 章

消息队列和 Celery

消息队列（Message Queue，简称 MQ）提供异步通信协议，可以实现进程间通信或同一进程的不同线程间通信。其中“消息”是指包含必要信息的数据。消息的发送者发送完毕后立即返回，消息被存储进队列中，对这个消息感兴趣的消费者会订阅消息并接收和处理它。使用消息队列的好处如下：

- 应用解耦。消息是平台无关和语言无关的，消息队列可以应对多变的产品变更。
- 异步通信。可以缩短请求等待的时间，使用专门处理请求的消费者来执行，提高 Web 页面的吞吐量，尤其是瞬间发生的高流量情况，消息队列非常有助于顶住访问压力。
- 数据持久化。未完成的消息不会因为某些故障而丢失。
- 送达保证。消息队列提供的冗余机制保证了消息确实能被处理。除非消费者明确表示已经处理完这个消息，否则这个消息可以被放回队列中以备其他消费者处理。

本章主要包含如下内容：

- 使用 Beanstalkd。
- 解释 AMQP，深入理解 RabbitMQ，介绍 RabbitMQ 插件系统，RabbitMQ 集群的故障转移方法等。
- 介绍 Celery 的架构，运行起一个真实的应用，在 Flask 应用中使用 Celery 等功能。
- 深入 Celery，介绍 Celery 的依赖及独立用法、Worker 管理、监控等高级功能。
- 笔者总结的一些实践经验。

使用 Beanstalkd

Beanstalkd 是消息队列的后起之秀，它是一个高性能、轻量级的分布式内存队列系统，最初设计的目的是想通过后台异步执行耗时的任务来降低 Web 应用的页面访问延迟。它支持过有 950 万用户的 Facebook Causes 应用，豆瓣也使用它作为消息队列。

Beanstalkd 有如下特点：

- 可持久化。Beanstalkd 运行使用内存，但也提供了持久性支持。在启动的时候使用 `-b` 参数指定持久化目录，它会将所有的任务写入 `binlog` 文件。在发生断电等情况后，用同样的参数指定重启它，将恢复 `binlog` 中的内容。
- 支持任务优先级。值越小优先级越高。
- 任务超时重发。消费者必须在预设的 TTR（Time To Run）时间内发送 `delete/release/bury` 改变任务状态，否则它会认为消息处理失败，把任务交给别的消费者节点执行。
- 支持任务预留。如果任务因为某些原因无法执行，消费者可以把任务置为 `buried` 状态保留这些任务。
- 支持分布式。客户端可以实现和 Memcached 一样的分布式。
- 灵活设置任务过期和 TTR 时间。

job 就是待异步执行的任务，也就是消息，是 Beanstalkd 中的基本单元。一个 job 通过生产者使用 `put` 命令时创建，然后被放在一个管道（tube）中。在整个生命周期中 job 可能有 4 个工作状态。

- `ready`：等待被取出并处理。
- `reserved`：如果 job 被消费者（worker）取出，将被此消费者预订，消费者将执行此 job。
- `delayed`：等待特定时间之后，状态再改为 `ready` 状态。
- `buried`：等待唤醒，通常在 job 处理失败时，会变成这个状态。

一个服务器有一个或者多个管道，管道用来存储统一类型的 job。每个管道由一个就绪队列与延迟队列组成。每个 job 所有的状态迁移在一个管道中完成。消费者监控/取消监控感兴趣的管道。当一个客户端连接上服务器时，客户端监控的管道默认为 `default`，如果客户端提交任务时没有使用 `use` 命令，那么这些 job 就存于名为 `default` 的管道中。管道都是按需求创建的，无论它们在什么时候被引用到。如果一个管道变为空也没有任何客户端引用，它将会被自动删除。

我们先安装它：

```
> git clone https://github.com/kr/beanstalkd
> cd beanstalkd
> sudo make install
```

adm 目录下包含一些管理服务的脚本。修改 beanstalkd.service：

```
[Unit]
Description=Beanstalkd is a simple, fast work queue

[Service]
User=ubuntu
ExecStart=/usr/local/bin/beanstalkd -b /opt/beanstalkd -p 11200
```

启动服务：

```
> sudo cp adm/systemd/beanstalkd.service /lib/systemd/system/
> sudo systemctl daemon-reload
> sudo systemctl start beanstalkd
```

使用 Beanstalkc

Beanstalkd 借鉴了 Memcached 设计，它们的协议和使用方式的风格很像。本节使用 Beanstalkd 的 Python 客户端 Beanstalkc (<http://bit.ly/28XuQYd>) 演示 Beanstalkd 的用法。

我们先安装它：

```
> pip install PyYAML beanstalkc # 使用PyYAML库可以让输出更直观
```

在交互模式下使用 Beanstalkc：

```
In : import beanstalkc
In : beanstalk = beanstalkc.Connection(host='localhost', port=11200)
In : beanstalk.tubes() # 列出全部管道
['default']
In : beanstalk.use('web_app') # 切换到管道web_app
Out: 'web_app'
In : beanstalk.watch('web_app') # 监控管道web_app
Out: 2
In : id = beanstalk.put('job_1', priority=21) # 放入一个任务，可以指定优先级
In : job = beanstalk.reserve() # 接收任务
In : job.body
Out: 'job_1'
In : job.stats() # 查看任务状态
Out:
```

```

{'age': 33,
 'buries': 0,
 'delay': 0,
 ...
 'tube': 'web_app'}
In : beanstalk.stats_tube('web_app') # 查看管道状态
Out:
{'cmd-delete': 0,
 'cmd-pause-tube': 0,
 'current-jobs-buried': 0,
 'current-jobs-delayed': 0,
 'current-jobs-ready': 0,
 'current-jobs-reserved': 1,
 ...
 'total-jobs': 1}
In : beanstalk.stats() # 查看链接状态
Out:
{'binlog-current-index': 3,
 ...
 'cmd-reserve': 3,
 'cmd-reserve-with-timeout': 0,
 'cmd-stats': 3,
 'cmd-stats-job': 7,
 'cmd-stats-tube': 3,
 ...
 'version': '1.10+21+gb7b4a6a'}
# 如果接收任务后没有修改任务状态, 任务完成后应该删除任务, 否则在TTR时间后会导致
  任务超时重发
In : job.delete()
In : beanstalk.close()

```

深入理解 RabbitMQ

RabbitMQ 是一个实现了 AMQP 协议标准的开源消息代理和队列服务器。和 Beanstalkd 不同的是, 它是企业级消息系统, 自带了集群、管理、插件系统等特性, 在高可用性、可扩展性、易用性等方面做得很好, 现在被互联网公司广泛应用。

我们先安装它:

```
> sudo apt-get install rabbitmq-server -yq
```

再安装 RabbitMQ 的 Python 客户端, 最常用的客户端是 pika:

```
> pip install pika
```

AMQP

AMQP (Advanced Message Queuing Protocol, 高级消息队列协议) 是一个异步消息传递所使用的应用层协议规范。它的设计初衷是为了摆脱商业 MQ 高额费用和不同 MQ 供应商的接口不统一的问题, 所以一开始就设计成开放标准, 以解决企业复杂的消息队列需求问题。

我们先了解几个概念。

1. 消息 (Message)。消息实际包含两部分内容:
 - 有效载荷 (Payload), 也就是要传输的数据, 数据类型可以纯文本也可以是 JSON。
 - 标签 (Label), 它包含交换机的名字和可选的主题 (topic) 标记等, AMQP 仅仅描述了标签, 而 RabbitMQ 决定了把这个消息发给哪个消费者。
2. 发布者 (Producer): 也就是生产者, 它创建消息并且设置标签。
3. 消费者 (Consumer): 消费者连接到代理服务器上, 接收消息的有效载荷 (注意, 消费者并不需要消息中的标签)。

AMQP 的工作流程如图 9.1 所示。

为了保证消息被正确取出并执行、消息投递失败后会重发, AMQP 模块包含了一个消息确认的概念: 当一个消息从队列中投递给消费者后, 消费者会通知消息代理 (也就是常说的 Broker), 这个通知可以是自动完成的, 也可以由处理消息的应用来执行。当消息确认 (Ack) 被启用的时候, 消息代理不会完全将消息从队列中删除, 除非收到来自消费者的确认回执。

交换机拿到一个消息之后会将它路由给队列。它使用哪种路由算法是由交换机类型和被称作 “绑定” (queue_bind) 的规则所决定的。目前 RabbitMQ 提供了如下四种交换机。

1. 直连交换机 (direct exchange): 根据消息携带的路由键 (routing key) 将消息投递给对应队列。将一个队列绑定到某个交换机的同时赋予该绑定一个路由键, 当一个携带着路由键为 XXX 的消息被发送给直连交换机时, 交换机会把它路由给绑定值同样为 XXX 的队列。直连交换机用来处理消息的单播路由。
2. 主题交换机 (topic exchange): 通过对消息的路由键和队列到交换机的绑定模式之间的匹配, 将消息路由给一个或多个队列。主题交换机通常用来实现消息的多播路由。发送到主题交换机的消息的路由键, 必须是一个由 “.” 分隔的词语列表, 这些词语应该和对应的业务相关联, 词语的个数可以随意, 但是不要超过 255 字节。绑定键支持通配符: “*” (星号) 用来表示一个单词; “#” (井号) 用来表示任意数量 (零个或多个) 单词。

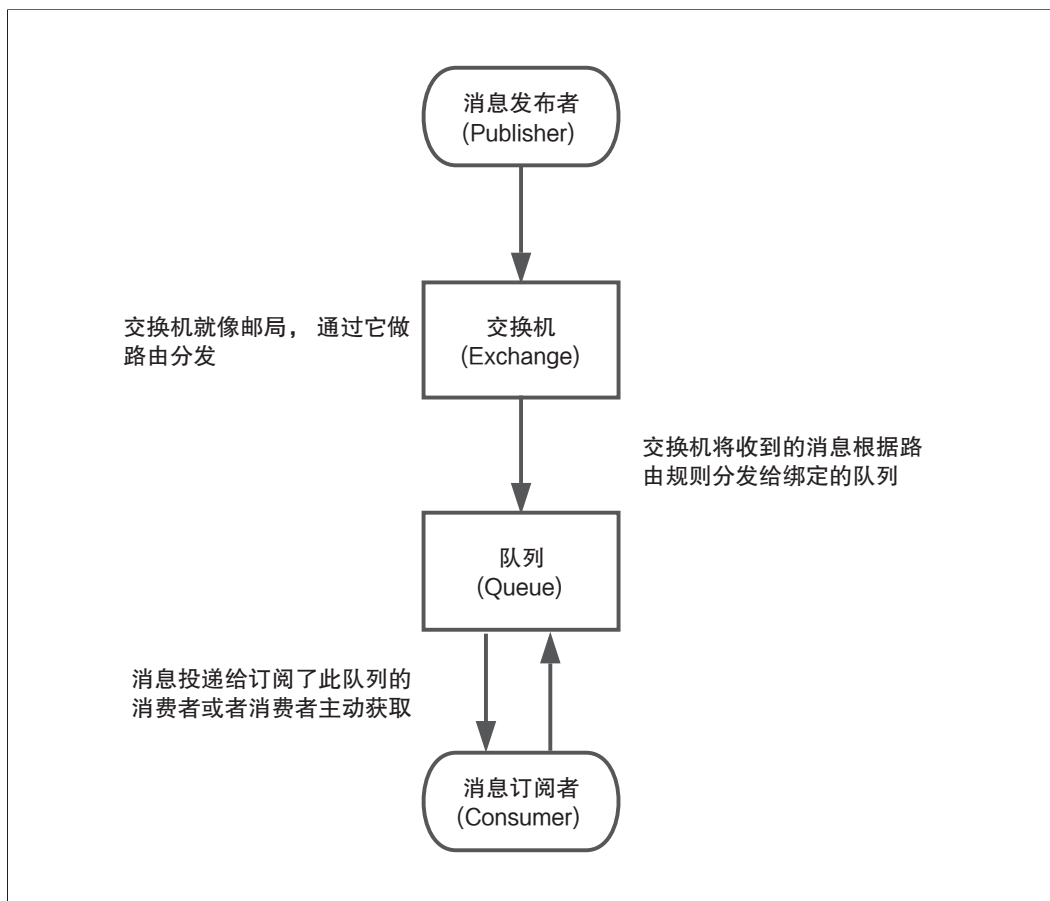


图 9.1 AMQP 的工作流程图

3. 扇型交换机 (fanout exchange)：将消息路由给绑定到它身上的所有队列，且不理睬绑定的路由键。扇型交换机用来处理消息的广播路由。扇型交换机非常有用，因为它允许你对单条消息做不同的处理，Web 开发中一个操作可能要做多个连带工作，比如用户创建一篇新的日记，需要更新用户的创建日记数、清除相关缓存、给关注这个用户的其他用户推消息、日记进审核后台、日记进最新日记池等等，可以使用扇型交换机把一个消息分发给多个任务队列，执行不一样的工作。尤其是当业务改变时，使用扇型交换机直接为新的消费者添加声明并绑定进来就可以了，否则需要修改发送方的代码来添加接收方。所以，使用扇型交换机可以有效地解耦发布者和消费者。
4. 头交换机 (headers exchange)：允许匹配 AMQP 的头而非路由键，其实使用起来和直接交换机差不多，但是性能却差很多，一般用不到这种类型。

下面通过一个完备的直接交换的例子，演示一下 RabbitMQ 的工作流程。首先看一下发布者的代码（amqp_producer.py）：

```
import sys

import pika

# %2F是被转义的“/”，这里使用了默认的虚拟主机和默认的用户及密码
parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')
connection = pika.BlockingConnection(parameters) # connection就是所谓的消息代理
channel = connection.channel() # 获得信道
# 声明交换机，指定交换类型为直接交换。最后两个参数表示想要持久化的交换机，其中
# durable为True表示RabbitMQ在崩溃重启之后会重建队列和交换机
channel.exchange_declare(exchange='web_develop', exchange_type='direct',
                        passive=False, durable=True, auto_delete=False)

if len(sys.argv) != 1:
    msg = sys.argv[1] # 使用命令行参数作为消息体
else:
    msg = 'hah'

# 创建一个消息，delivery_mode为2表示让这个消息持久化，重启RabbitMQ也不会丢失。使
# 用持久化需要考虑为此付出的性能成本，如果开启此功能，强烈建议把消息存储在SSD
# 上
props = pika.BasicProperties(content_type='text/plain', delivery_mode=2)
# basic_publish表示发送路由键为xxx_routing_key，消息体为haha的消息给web_develop这个
# 交换机
channel.basic_publish('web_develop', 'xxx_routing_key', msg,
                    properties=props)
connection.close() # 关闭连接
```

上述例子使用了文本模式（text/plain），复杂的内容类型还可以选择 JSON 格式：application/json。信道是建立在 TCP 连接内的虚拟连接。不直接通过 TCP 连接发送 AMQP 命令，是因为对操作系统而言，创建和销毁 TCP 连接是非常昂贵的开销，而且 TCP 连接数量是有限，很容易出现性能瓶颈，一个 TCP 连接上的信道数量没有限制，性能也非常好。

再看一下消费者的代码（amqp_consumer.py）：

```
import pika

# 处理接收到的消息的回调函数
# method_frame携带了投递标记，header_frame表示AMQP信息头的对象
# body为消息实体
def on_message(channel, method_frame, header_frame, body):
    # 消息确认，确认之后才会删除消息并给消费者发送新的消息
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)
    print body
```

```

parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')
connection = pika.BlockingConnection(parameters)
channel = connection.channel()

channel.exchange_declare(exchange='web_develop', exchange_type='direct',
                        passive=False, durable=True, auto_delete=False)
# 声明队列, 如果没有就创建
channel.queue_declare(queue='standard', auto_delete=True)
# 通过路由键将队列和交换机绑定
channel.queue_bind(queue='standard', exchange='web_develop',
                  routing_key='xxx_routing_key')

channel.basic_consume(on_message, 'standard') # 订阅队列

try:
    channel.start_consuming() # 开始消费
except KeyboardInterrupt:
    channel.stop_consuming() # 停止消费

connection.close()

```

现在先启动消费者:

```
> python chapter9/section2/amqp_consumer.py
```

然后使用另外一个终端, 启动发布者程序发送消息:

```
> python chapter9/section2/amqp_producer.py web
```

执行完之后就可以在消费者的终端上看到“web”了。

现在对发布者的例子加以改动, 让它支持消息确认。支持的原理是确保 basic_publish 的返回值为 True (mqp_producer_with_confirm.py):

```

...
channel = connection.channel()
# 接收确认消息
channel.confirm_delivery()
if channel.basic_publish('web_develop', 'xxx_routing_key', msg,
                        properties=props):
    print 'Message publish was confirmed!'
else:
    print 'Message could not be confirmed!'
connection.close()

```

虚拟主机

RabbitMQ 服务器可以创建虚拟主机，它能拥有自己的队列、绑定和交换机，就像一个有自己的权限机制的迷你版 RabbitMQ。不同的虚拟主机之间完全隔离，还可以有效避免命名冲突的问题。上面的例子都基于默认的虚拟主机 “/”，虚拟主机需要在连接的时候指定。肯定不能在生产环境使用 `guest` 这个默认的用户/密码。接下来我们来创建自己的用户和虚拟机，并给它赋予一定的权限。

管理 RabbitMQ 一般都是通过 `rabbitmqctl` 这个命令来完成的：

```
> sudo rabbitmqctl add_user dongwm 123456
> sudo rabbitmqctl add_vhost web_develop
> sudo rabbitmqctl set_permissions -p web_develop dongwm ".*" ".*" ".*"
```

其中 `set_permissions` 后面的三个 “.*”，分别是配置（队列和交换的创建和删除）、写（发布消息）、读（消费消息）的权限。

看一下目前存在的虚拟主机、队列和用户：

```
> sudo rabbitmqctl list_vhosts
Listing vhosts ...
/
web_develop
> sudo rabbitmqctl list_queues -p web_develop # 还没有声明，所以没有队列
Listing queues ...
> sudo rabbitmqctl list_users
Listing users ...
dongwm []
guest [administrator]
```



默认的用户 `guest` 应该在线前删除，取消管理员权限或者改变密码。

插件系统

RabbitMQ 提供了强大的插件系统，当你需要某些功能而 RabbitMQ 没有时，可以在网络上查找或者自己编写一个插件安装进来使用。使用插件可以实现如下功能：

- 管理和监控 RabbitMQ。
- 支持 AMQP 之外的协议。

- 消息复制。
- 新的路由算法和交换类型。

官方的插件列表页 (<https://www.rabbitmq.com/plugins.html>) 列出了多个插件, 其中 RabbitMQ 团队维护的插件都放在支持的插件 (Supported Plugins) 列表中, 这些插件会和 RabbitMQ 最新版保持一致; 而实验性质的插件 (Experimental Plugins) 列表建议不要在生产环境中使用。

安装插件的方法非常简单:

```
> sudo rabbitmq-plugins enable rabbitmq_shovel
```

rabbitmq_shovel 插件就安装好了, Shovel 用于有多个数据中心的环境, 能够定义 RabbitMQ 上的队列和另一个 RabbitMQ 上的交换机的复制关系, 这样消息就可以被异步复制到其他的数据中心并被消费了。

对应的移除插件的命令如下:

```
> sudo rabbitmq-plugins disable rabbitmq_shovel
```

通过 Web 和 REST API 管理 RabbitMQ

有的人喜欢用命令行的方式管理, 有的人喜欢在图形界面上管理, RabbitMQ 提供 Web 管理程序, 但是这个功能是通过 rabbitmq_management 实现的, 需要安装它:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Web 服务默认的端口是 15672, 虚拟机没有添加这个端口的端口转发, 所以需要修改配置, 把端口换成 5000:

```
> cat /etc/rabbitmq/rabbitmq.config
[
  {rabbitmq_management, [{listener, [{port, 5000}]}]}
].
```

这个配置采用的是 Erlang 的数据结构, 看起来是一个包含了嵌套字典的数组, 但是要注意结尾的点。重启 rabbitmq-server:

```
> sudo /etc/init.d/rabbitmq-server restart
```

登录 Web 界面之前还需要给用户添加权限, 管理插件时有 5 种权限, 如表 9.1 所示。

表 9.1 管理插件时的 5 种权限

权 限	含 义
None	也就是什么都不做，新创建的用户默认没有登录管理页面的权限
management	查看用户有权限访问的虚拟主机的队列、交换机、绑定、通道和连接等
polycymaker	除了 management 的权限外，还能查看、创建和删除策略和参数
monitoring	除了 management 的权限外，还能查看其他用户的通道和连接、列出全部虚拟主机等
administrator	最高权限

使用 administrator 权限：

```
> sudo rabbitmqctl set_user_tags dongwm administrator
```

现在就可以使用刚才创建的 dongwm 这个用户登录管理页面 <http://localhost:5000> 了，登录之后的页面如图 9.2 所示。

这个 Web 界面支持如下操作：

- 服务器数据和统计预览。如最近一段时间的队列情况、当前连接数、当前队列数、内存占用、RabbitMQ 版本、主机名等。
- 导出/导入服务器配置。
- 查看服务器连接。
- 查看信道列表。
- 查看交换机列表，添加新的交换机。
- 查看队列，添加新的队列，修改队列绑定。
- 查看用户列表，添加用户。
- 查看虚拟主机，添加新的虚拟主机。
- 列出策略，添加/更新策略。

rabbitmq_management 插件还提供了一个 HTTP 的 REST API 服务，可以使用 curl 或者 httpie 这样的工具执行下面的操作：

```
> http -a dongwm:123456 http://localhost:5000/api/nodes/rabbit@WEB
HTTP/1.1 200 OK
...
Server: MochiWeb/1.1 WebMachine/1.10.0 (never breaks eye contact)

{
  ...
}
```

```
"log_file": "/var/log/rabbitmq/rabbit@WEB.log",
"mem_alarm": false,
"mem_limit": 628290355,
"mem_used": 54851512,
"mem_used_details": {
  "rate": -1353.6
},
...
}
```

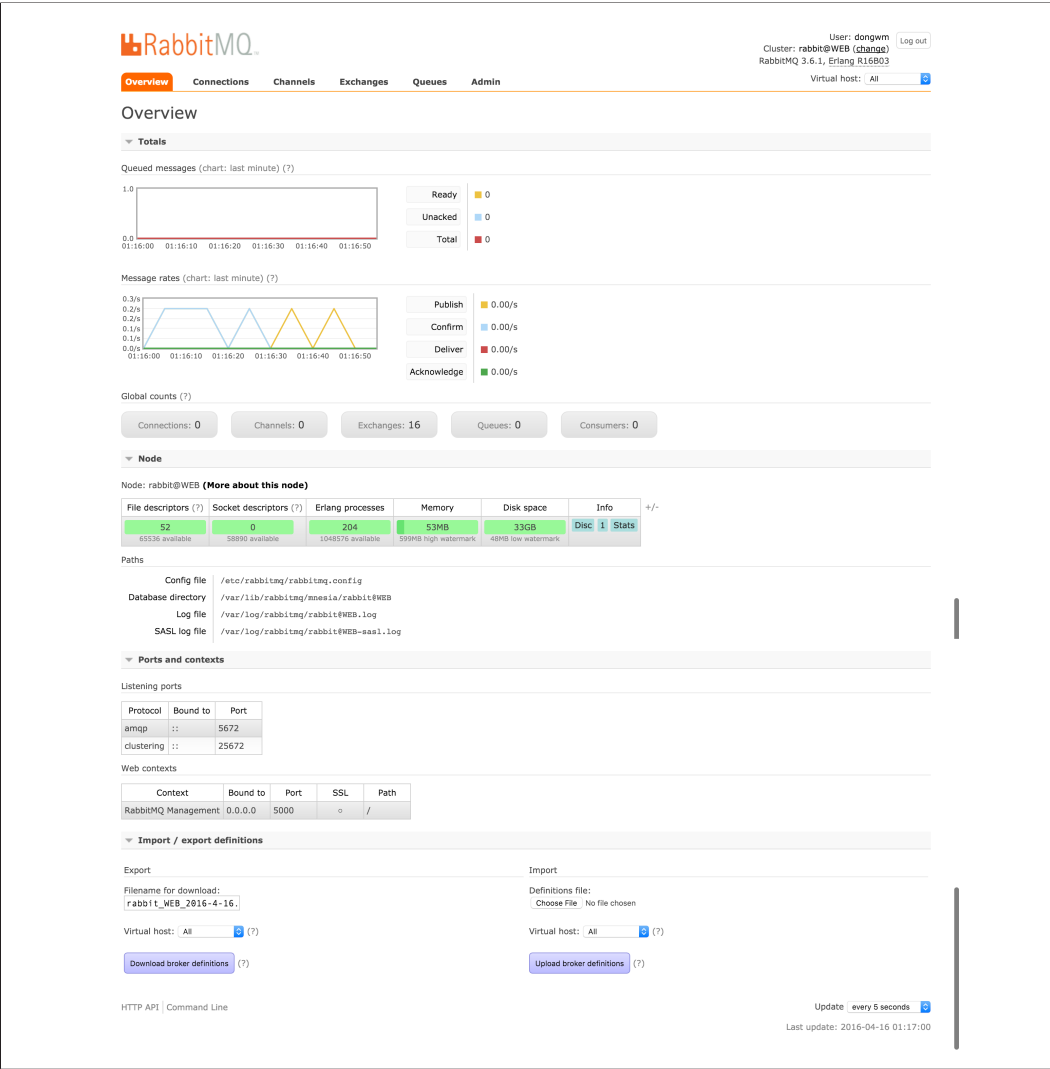


图 9.2 登录之后的页面

其他用法可以参考文档 <http://localhost:5000/api>。

还有一个非常有用的工具：`rabbitmqadmin`，它是一个使用 Python 编写的脚本，支持上述全部操作，我们可以方便地把它集成到运维监控和应用分析中。它需要通过 Web 页面下载：

```
> wget http://localhost:5000/cli/rabbitmqadmin
> chmod +x rabbitmqadmin
```

看一下效果：

```
> sudo ./rabbitmqadmin -P 5000 list vhosts
+-----+-----+
|  name      | messages |
+-----+-----+
| /          | 0         |
| web_develop | 0         |
+-----+-----+
> sudo ./rabbitmqadmin -P 5000 purge queue name='standard'
queue purged
```

故障转移

一般使用 HAProxy 做 RabbitMQ 集群的负载均衡工具，当集群节点出现故障时，由应用程序决定故障转移的方式，最简单的方案就是让消费者重试：

```
while 1:
    try:
        connection = pika.BlockingConnection(parameters)
    except:
        pass
```

使用 Celery

Celery 是一个专注于实时处理和任务调度的分布式任务队列。所谓任务就是消息，消息中的有效载荷中包含要执行任务需要的全部数据。

使用 Celery 的常见场景如下：

- Web 应用。当用户触发的一个操作需要较长时间才能执行完成时，可以把它作为任务交给 Celery 去异步执行，执行完再返回给用户。这段时间用户不需要等待，提高了网站的整体吞吐量和响应时间。

- 定时任务。生产环境经常会跑一些定时任务。假如你有上千台的服务器、上千种任务，定时任务的管理很困难，Celery 可以帮助我们快速在不同的机器设定不同种任务。
- 其他可以异步执行的任务。为了充分提高网站性能，对于请求和响应之外的那些不要求必须同步完成的附加工作都可以异步完成。比如发送短信/邮件、推送消息、清理/设置缓存等。

Celery 还提供了如下的特性：

- 方便地查看定时任务的执行情况，比如执行是否成功、当前状态、执行任务花费的时间等。
- 可以使用功能齐备的管理后台或者命令行添加、更新、删除任务。
- 方便把任务和配置管理相关联。
- 可选多进程、Eventlet 和 Gevent 三种模式并发执行。
- 提供错误处理机制。
- 提供多种任务原语，方便实现任务分组、拆分和调用链。
- 支持多种消息代理和存储后端。

Celery 的架构

Celery 包含如下组件。

- Celery Beat：任务调度器，Beat 进程会读取配置文件的内容，周期性地将配置中到期需要执行的任务发送给任务队列。
- Celery Worker：执行任务的消费者，通常会在多台服务器运行多个消费者来提高执行效率。
- Broker：消息代理，或者叫作消息中间件，接受任务生产者发送过来的任务消息，存进队列再按序分发给任务消费方（通常是消息队列或者数据库）。
- Producer：调用了 Celery 提供的 API、函数或者装饰器而产生任务并交给任务队列处理的都是任务生产者。
- Result Backend：任务处理完后保存状态信息和结果，以供查询。Celery 默认已支持 Redis、RabbitMQ、MongoDB、Django ORM、SQLAlchemy 等方式。

Celery 的架构图如图 9.3 所示。

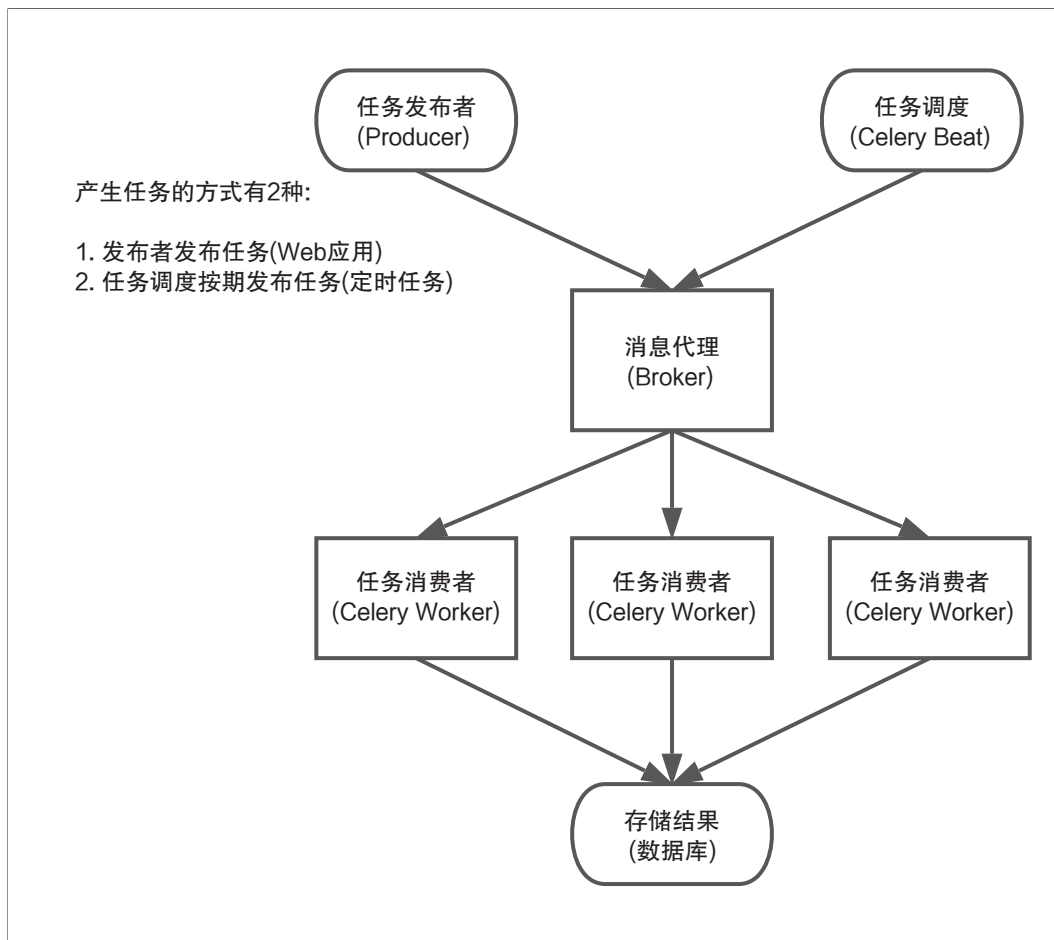


图 9.3 Celery 的架构图

选择消息代理

Celery 目前支持 RabbitMQ、Redis、MongoDB、Beanstalk、SQLAlchemy、Zookeeper 等作为消息代理，但适用于生产环境的只有 RabbitMQ 和 Redis，至于其他方式，一是支持有限，二是可能得不到更好的技术支持。

Celery 官方推荐的是 RabbitMQ，Celery 的作者 Ask Solem Hoel 最初在 VMware 就是为 RabbitMQ 工作的，Celery 最初的设计就是基于 RabbitMQ，所以使用 RabbitMQ 会非常稳定，成功案例很多。如果使用 Redis，则需要能接受发生突然断电之类的问题造成 Redis 突然终止后的数据丢失等后果。

Celery 序列化

在客户端和消费者之间传输数据需要序列化和反序列化，Celery 支持如表 9.2 所示的序列化方案。

表 9.2 Celery 支持的序列化方案

方 案	说 明
pickle	pickle 是 Python 标准库中的一个模块，支持 Python 内置的数据结构，但是它是 Python 的专有协议。从 Celery 3.2 开始，由于安全性等原因 Celery 将拒绝 pickle 这个方案
json	json 支持多种语言，可用于跨语言方案
yaml	yaml 的表达能力更强，支持的数据类型比 json 多，但是 Python 客户端的性能不如 JSON
msgpack	msgpack 是一个二进制的类 json 的序列化方案，但是比 json 的数据结构更小、更快

安装配置 Celery

为了提供更高的性能，我们选择如下方案：

- 选择 RabbitMQ 作为消息代理。
- RabbitMQ 的 Python 客户端选择 librabbitmq 这个 C 库。
- 选择 Msgpack 做序列化。
- 选择 Redis 做结果存储。

下面先安装它们。Celery 提供 bundles 的方式，也就是安装 Celery 的同时可以一起安装多种依赖：

```
> pip install "celery[librabbitmq,redis,msgpack]"
```



bundles 的原理是在 setup.py 的 setup 函数中添加 extras_require。

从一个简单的例子开始

先演示一个简单的项目让 Celery 运行起来。项目的目录结构如下：

```
> tree chapter9/section3/proj
├── celeryconfig.py
├── celery.py
├── __init__.py
└── tasks.py
```

先看一下主程序 `celery.py`:

```
from __future__ import absolute_import

from celery import Celery

app = Celery('proj', include=['proj.tasks'])
app.config_from_object('proj.celeryconfig')

if __name__ == '__main__':
    app.start()
```

解析一下这个程序:

- “`from __future__ import absolute_import`” 是拒绝隐式引入, 因为 `celery.py` 的名字和 `celery` 的包名冲突, 需要使用这条语句让程序正确地运行。
- `app` 是 `Celery` 类的实例, 创建的时候添加了 `proj.tasks` 这个模块, 也就是包含了 `proj/tasks.py` 这个文件。
- 把 `Celery` 配置存放在 `proj/celeryconfig.py` 文件, 使用 `app.config_from_object` 加载配置。

看一下存放任务函数的文件 `tasks.py`:

```
from __future__ import absolute_import

from proj.celery import app

@app.task
def add(x, y):
    return x + y
```

`tasks.py` 只有一个任务函数 `add`, 让它生效的最直接的方法就是添加 `app.task` 这个装饰器。

看一下我们的配置文件 `celeryconfig.py`:

```
BROKER_URL = 'amqp://dongwm:123456@localhost:5672/web_develop' # 使用RabbitMQ作为消息代理
CELERY_RESULT_BACKEND = 'redis://localhost:6379/0' # 把任务结果存在了Redis
CELERY_TASK_SERIALIZER = 'msgpack' # 任务序列化和反序列化使用msgpack方案
```

```
CELERY_RESULT_SERIALIZER = 'json' # 读取任务结果一般性能要求不高, 所以使用了可读
性更好的JSON
CELERY_TASK_RESULT_EXPIRES = 60 * 60 * 24 # 任务过期时间, 不建议直接写86400, 应该让
这样的magic数字表述更明显
CELERY_ACCEPT_CONTENT = ['json', 'msgpack'] # 指定接受的内容类型
```

这个例子中没有任务调度相关的内容, 所以只需要启动消费者:

```
> cd ~/web_develop/chapter9/section3
> celery -A proj worker -l info
```



-A 参数默认会寻找 proj.celery 这个模块, 其实使用 celery 作为模块文件名字不怎么合理。可以使用其他名字。举个例子, 假如是 proj/app.py, 可以使用如下命令启动:

```
> celery -A proj.app worker -l info
```

如果看到如下的启动信息, 就说明 worker 服务运行起来了:

```
----- celery@WEB v3.1.18 (Cipater)
---- **** ----
-- * *** * -- Linux-4.4.0-22-generic-x86_64-with-Ubuntu-16.04-xenial
-- * _ **** --
- ** ----- [config]
- ** ----- .> app:          proj:0x7f444c1b01d0
- ** ----- .> transport:   amqp://dongwm:**@localhost:5672/web_develop
- ** ----- .> results:    redis://localhost:6379/0
- *** --- * --- .> concurrency: 1 (prefork)
-- ***** ---
--- ***** ----- [queues]
----- .> celery          exchange=celery(direct) key=celery
```

[tasks]

```
. proj.tasks.add
```

```
[2016-06-03 13:22:50,368: INFO/MainProcess] Connected to amqp://dongwm:**@localhost
:5672/web_develop
[2016-06-03 13:22:50,381: INFO/MainProcess] mingle: searching for neighbors
[2016-06-03 13:22:51,392: INFO/MainProcess] mingle: all alone
[2016-06-03 13:22:51,629: WARNING/MainProcess] celery@WEB ready.
```

上述信息提供了一些有帮助的内容, 如消息代理和存储结果的地址、并发数量、任务列表、交换类型等。在对 Celery 不熟悉的时候可以通过如上信息判断设置和修改是否已生效。

现在开启另外一个终端，用 IPython 调用 add 函数：

```
In : from proj.tasks import add
In : r = add.delay(1, 3)
In : r
Out: <AsyncResult: 93288a00-94ee-4727-b815-53dc3474cf3f>
In : r.result
Out: 4
In : r.status
Out: u'SUCCESS'
In : r.successful()
Out: True
In : r.backend
Out: <celery.backends.redis.RedisBackend at 0x7fb2529500d0> # 保存在Redis中
```

可以看到 worker 的终端上显示执行了任务：

```
[2016-06-03 13:34:40,749: INFO/MainProcess] Received task: proj.tasks.add[93288a00-94ee-4727-b815-53dc3474cf3f]
[2016-06-03 13:34:40,755: INFO/MainProcess] Task proj.tasks.add[93288a00-94ee-4727-b815-53dc3474cf3f] succeeded in 0.00511166098295s: 4
```

通过 IPython 触发的任务就完成了。任务的结果都需要根据上面提到的 task_id 获得，我们还可以用如下两种方式随时找到这个结果：

```
task_id = '93288a00-94ee-4727-b815-53dc3474cf3f'
In : add.AsyncResult(task_id).get()
Out: 4
```

或者：

```
In : from celery.result import AsyncResult
In : AsyncResult(task_id).get()
Out: 4
```

指定队列

Celery 非常容易设置和运行，通常它会使用默认的名为 celery 的队列（可以通过 CELERY_DEFAULT_QUEUE 修改）用来存放任务。我们可以使用优先级不同的队列来确保高优先级的任务不需要等待就得到响应。

基于 proj 目录下的源码，我们创建一个 projq 目录，并对 projq/celeryconfig.py 添加如下配置：

```
from kombu import Queue

CELERY_QUEUES = ( # 定义任务队列
```

```

    Queue('default', routing_key='task.#'), # 路由键以“task.”开头的消息都进
        default队列
    Queue('web_tasks', routing_key='web.#'), # 路由键以“web.”开头的消息都进
        web_tasks队列
)
CELERY_DEFAULT_EXCHANGE = 'tasks' # 默认的交换机名字为tasks
CELERY_DEFAULT_EXCHANGE_TYPE = 'topic' # 默认的交换类型是topic
CELERY_DEFAULT_ROUTING_KEY = 'task.default' # 默认的路由键是task.default, 这个路由
    键符合上面的default队列

CELERY_ROUTES = {
    'projq.tasks.add': { # tasks.add的消息会进入web_tasks队列
        'queue': 'web_tasks',
        'routing_key': 'web.add',
    }
}

```

现在用指定队列的方式启动消费者进程:

```
> celery -A projq worker -Q web_tasks -l info
```

上述 worker 只会执行 web_tasks 中的任务, 我们可以合理安排消费者数量, 让 web_tasks 中任务的优先级更高。

使用任务调度

之前的例子都是由发布者触发的, 本节展示一下使用 Celery 的 Beat 进程自动生成任务。基于 proj 目录下的源码, 创建一个 projb 目录, 对 projb/celeryconfig.py 添加如下配置:

```

CELERYBEAT_SCHEDULE = {
    'add': {
        'task': 'projb.tasks.add',
        'schedule': timedelta(seconds=10),
        'args': (16, 16)
    }
}

```

CELERYBEAT_SCHEDULE 中指定了 tasks.add 这个任务每 10 秒跑一次, 执行的时候的参数是 16 和 16。

启动 Beat 程序:

```
> celery beat -A projb
```

然后启动 Worker 进程：

```
> celery -A projb worker -l info
```

之后可以看到每 10 秒都会自动执行一次 tasks.add。



Beat 和 Worker 进程可以一并启动：

```
> celery -B -A projb worker -l info
```

使用 Django 可以通过 django-celery 实现在管理后台创建、删除、更新任务，是因为它使用了自定义的调度类 `djcelery.schedulers.DatabaseScheduler`，我们可以参考它实现 Flask 或者其他 Web 框架的管理后台来完成同样的功能。使用自定义调度类还可以实现动态添加任务。

任务绑定、记录日志和重试

任务绑定、记录日志和重试是 Celery 常用的 3 个高级属性。现在修改 `proj/tasks.py` 文件，添加 `div` 函数用于演示：

```
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@app.task(bind=True)
def div(self, x, y):
    logger.info(('Executing task id {0.id}, args: {0.args!r} '
                'kwargs: {0.kwargs!r}').format(self.request))
    try:
        result = x / y
    except ZeroDivisionError as e:
        raise self.retry(exc=e, countdown=5, max_retries=3)
    return result
```

当使用 `bind = True` 后，函数的参数发生变化，多出了参数 `self`（第一个参数），相当于把 `div` 变成了一个已绑定的方法，通过 `self` 可以获得任务的上下文。

在 IPython 中调用 `div`：

```
In : from proj.tasks import div
In : r = div.delay(2, 1)
```


可以看到如下执行信息：

```
[2016-06-03 15:50:31,853: INFO/Worker-1] proj.tasks.div[1da82fb8-20de-4d5a-9b48-045da6db0cda]: Executing task id 1da82fb8-20de-4d5a-9b48-045da6db0cda, args: [2, 1]
kwargs: {}
```

换成能造成异常的参数：

```
In : r = div.delay(2, 0)
```

可以发现每 5 秒就会重试一次，一共重试 3 次（默认重复 3 次），然后抛出异常。

在 Flask 应用中使用 Celery

在 Web 应用中，用户请求页面发布任务，交由 Celery 后端处理。一种是把任务发布然后请求继续进行，响应不需要获知任务的执行情况；另外一种是需要实时把任务的执行过程反馈到用户的浏览器上。本节将演示这两种任务的处理方式。

Socket.IO 是一个支持 WebSocket 协议，面向实时 Web 应用的 JavaScript 库。实现了浏览器与服务器之间的双向通信。它有两个部分：

- 在浏览器中运行的客户端库。这个库由 Socket.IO 官方提供。
- Python 实现的服务端库。

为了简化代码，我们使用 Flask-SocketIO 这个扩展，首先安装它：

```
> pip install Flask-SocketIO
```

Celery 默认使用多进程的方式运行 Worker 进程，这个例子将使用 eventlet 的方法运行 Worker 以及 SocketIO。首先需要引入依赖包和设置（celery_socketio.py）：

```
from flask import Flask, render_template
from flask_socketio import SocketIO
from celery import Celery
import eventlet
eventlet.monkey_patch()

app = Flask(__name__)
here = os.path.abspath(os.path.dirname(__file__)) # 获取文件的绝对目录路径
app.config.from_pyfile(os.path.join(here, 'proj/celeryconfig.py'))

SOCKETIO_REDIS_URL = app.config['CELERY_RESULT_BACKEND']
socketio = SocketIO(
    app, async_mode='eventlet',
    message_queue=SOCKETIO_REDIS_URL) # 使用Redis存储SocketIO的消息队列
```

```
celery = Celery(app.name)
celery.conf.update(app.config)
```

接下来编写 Celery 任务：

```
@celery.task
def background_task():
    socketio.emit( # emit可以理解为向浏览器发送数据
        'my response', {'data': 'Task starting ...'},
        namespace='/task')
    time.sleep(10)
    socketio.emit(
        'my response', {'data': 'Task complete!'},
        namespace='/task')
```

```
@celery.task
def async_task():
    print 'Async!'
    time.sleep(5)
```

然后编写视图函数：

```
@app.route('/')
def index():
    return render_template('chapter9/section3/index.html')

@app.route('/async')
def async():
    async_task.delay() # 访问/async会异步地发布一个async_task任务， 然后马上返回'
                        Task complete!'
    return 'Task complete!'

@app.route('/task')
def start_background_task():
    background_task.delay()
    return 'Started'
```

最后，我们设置启动应用的方式，和之前的 Flask 用法略有不同：

```
if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=9000, debug=True)
```

创建一个简单的模板 `index.html`，它带有很少的样式，用 jQuery 操作 DOM。为了减少依赖，`socket.io.js` 引用了外部 CDN 的地址：

```
<script type="text/javascript" src="//cdn.socket.io/socket.io-1.4.5.js"></script>
```

先看一下模板的主体：

```
<body>
  <h3>Logging</h3>
  <p id="log"></p>
  <button id="background">Execute</button>
</body>
```

再看一下 JavaScript 事件部分：

```
<script type="text/javascript">
  $(document).ready(function(){
    namespace = '/task';
    socket = io.connect('http://' + document.domain + ':' + location.port +
      namespace);
    socket.on('my response', function(msg) {
      console.log('Received: ' + msg.data);
      $('#log').append('Received: ' + msg.data + '<br>');
    });
    $('#background').on('click', function() {
      $.get("{{ url_for('start_background_task') }}");
    });
  });
</script>
```

上面这段代码表示当页面加载完成之后，我们会创建 Websocket 链接，并且给 id 为 `background` 的 DOM 添加一个只要点击就会访问视图函数名为 `start_background_task`（也就是访问 `/task`）的事件。

现在运行 Flask 应用和 Celery：

```
> cd ~/web_develop/chapter9/section3
> python celery_socketio.py
> celery -A celery_socketio.celery -P eventlet worker -l info
```

访问 “`http://localhost:9000/async`” 可以感受到这个请求是直接返回的，`sleep 5` 秒的操作并没有影响到页面响应时间。

访问首页 “`http://localhost:9000/`”，单击 `Execute` 按钮，可以看到红色框里面的内容会更新，执行完毕后如图 9.4 所示。



图 9.4 红色框里面的内容会更新

深入理解 Celery

Celery 的依赖

Celery 依赖了相关的三个库：billiard、librabbitmq 和 kombu，这些库都由 Celery 的作者开发和维护。

billiard

billiard 是一个基于 Python 2.7 标准库多进程模块 multiprocessing 而改进的库，它主要用来提高性能和稳定性，也可以兼容还没有 multiprocessing 模块的早期 Python 版本。Celery 消费者的多进程模式称为 prefork。

librabbitmq

librabbitmq 是一个 C 语言实现的 Python 客户端。虽然之前我们介绍 RabbitMQ 的时候使用了 pika，而 Celery 也实现了 py-amqp 这个 Python 客户端，但是基于性能原因，请务必使用 librabbitmq，只有当 librabbitmq 不可用的时候才使用 py-amqp（和 librabbitmq 的接口一致）。

我们来把之前的 pika 的例子移植成使用 librabbitmq。先看发布者（librabbitmq_producer.py）：

```
import sys

from librabbitmq import Connection

connection = Connection(host='localhost', userid='dongwm',
                        password='123456', virtual_host='web_develop')
channel = connection.channel()

channel.exchange_declare('web_develop', 'direct',
                        passive=False, durable=True, auto_delete=False)

if len(sys.argv) != 1:
    msg = sys.argv[1]
```

```
else:
    msg = 'hah'

channel.basic_publish(msg, 'web_develop', 'xxx_routing_key')

connection.close()

再看消费者 (librabbitmq_consumer.py):

from librabbitmq import Connection

connection = Connection(host='localhost', userid='dongwm',
                        password='123456', virtual_host='web_develop')

channel = connection.channel()

def on_message(message):
    print("Body:%s", Properties:%s", DeliveryInfo:%s" % (
        message.body, message.properties, message.delivery_info))
    message.ack()

channel.exchange_declare('web_develop', 'direct',
                        passive=False, durable=True, auto_delete=False)

channel.queue_declare('standard', auto_delete=True)
channel.queue_bind('standard', 'web_develop', 'xxx_routing_key')
channel.basic_consume('standard', callback=on_message)

try:
    while True:
        connection.drain_events()
except KeyboardInterrupt:
    exit(1)
```

kombu

kombu 是 Celery 自带的用来收发消息的库，它提供了符合 Python 语言习惯的、使用 AMQP 协议的高级接口。OpenStack 等项目都在使用它，原因如下：

- 支持非常多的消息代理。
- 支持对有效负荷数据的自动编码、序列化和压缩。

- 优雅地处理连接和通道错误。
- 提供多消息代理的一致性接口和异常处理方式，切换消息代理的代价很小。

把 pika 的例子替换为 kombu 的版本。先看发布者 (kombu_producer.py):

```
import sys

from kombu import Connection, Exchange, Queue, Producer

web_exchange = Exchange('web_develop', 'direct', durable=True)
standard_queue = Queue('standard', exchange=web_exchange,
                       routing_key='web.develop')
URI = 'librabbitmq://dongwm:123456@localhost:5672/web_develop'

if len(sys.argv) != 1:
    msg = sys.argv[1]
else:
    msg = 'hah'

with Connection(URI) as connection:
    producer = Producer(connection)
    producer.publish(
        msg, exchange=web_exchange, declare=[standard_queue],
        routing_key='web.develop',
        serializer='json', compression='zlib')
```

然后是消费者 (kombu_consumer.py):

```
from kombu import Connection, Exchange, Queue, Consumer
from kombu.async import Hub

web_exchange = Exchange('web_develop', 'direct', durable=True)
standard_queue = Queue('standard', exchange=web_exchange,
                       routing_key='web.develop')

URI = 'librabbitmq://dongwm:123456@localhost:5672/web_develop'
hub = Hub()

def on_message(body, message):
    print("Body:%s", Properties:%s", DeliveryInfo:%s" % (
        body, message.properties, message.delivery_info))
    message.ack()

with Connection(URI) as connection:
    connection.register_with_event_loop(hub)
```

```
with Consumer(connection, standard_queue, callbacks=[on_message]):
    try:
        hub.run_forever()
    except KeyboardInterrupt:
        exit(1)
```

任务调用

之前我们调用任务都使用 `delay` 方法，格式如下：

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

`delay` 其实是 `apply_async` 的别名，还可以使用如下方法调用：

```
task.apply_async(args=[arg1, arg2], kwargs={'kwarg1': 'x', 'kwarg2': 'y'})
```

二者的区别在于参数的表达方式，而且 `apply_async` 支持更多的参数。`apply_async` 支持的参数如下所述。

- 1. `countdown`：等待一段时间再执行。
`add.apply_async((2, 2), countdown=5)` # 5秒后再执行
- 2. `eta`：ETA 是 `estimated time of arrival` 的缩写，也就是定义了任务的开始时间。`countdown` 相当于特殊的单位为秒的 `eta`。
`add.apply_async((2, 2), eta=now + timedelta(seconds=10))` # 时间到了 `now` 加 10 秒，任务才开始
- 3. `expires`：设置超时时间。
`add.apply_async((2, 2), expires=60)` # 60 秒之后过期
- 4. `retry`：定义如果任务失败是否重试。
`add.apply_async((2, 2), retry=False)` # 失败了不会重试
- 5. `retry_policy`：重试策略。可以设置如表 9.3 所示的几个键。

表 9.3 可以设置的键及其含义

键	含 义
<code>max_retries</code>	最大重试次数，默认为 3 次
<code>interval_start</code>	重试等待的间隔秒数，默认为 0，表示直接重试不等待
<code>interval_step</code>	每次重试让重试间隔增加的秒数，比如第 1 次重试间隔 0 秒，第 2 次重试的间隔就是 0.2 秒……可以是数字或者浮点数，默认值是 0.2

续表

键	含 义
interval_max	重试间隔最大的秒数。也就是通过 interval_step 增大到多少秒之后就不再增加了。 可以是数字或者浮点数，默认值是 0.2

使用 `apply_async` 也可以自定义发布者、交换机、路由键、队列、优先级（目前只支持 Redis 和 Beanstalk）、序列方案和压缩方法：

```
add.apply_async((2, 2), compression='zlib',
                serializer='json', queue='priority.high',
                routing_key='web.add', priority=0, exchange='web_exchange')
```

信号系统

Celery 支持 7 种信号类型：

- 1. 任务信号。这个类型的信号最常用到，任务相关的信号有 8 种，如表 9.4 所示。

表 9.4 任务相关的信号及其含义

信 号	含 义
before_task_publish	任务发布前
after_task_publish	任务发布后
task_prerun	任务执行前
task_postrun	任务执行后
task_retry	任务重试时
task_success	任务完成时
task_failure	任务失败时
task_revoked	任务被撤销或者终止时

- 2. 应用信号。
- 3. Worker 信号。
- 4. Beat 信号。
- 5. Eventlet 信号。
- 6. 日志信号。

7. 命令信号。

不同的信号参数格式不同，具体的格式可以参考官方文档（<http://docs.celeryproject.org/en/latest/userguide/signals.html>）。现在将 9.3 节的 proj 复制到 chapter9/section4 目录下，给 proj/celery.py 添加如下一段订阅 after_task_publish 信号的代码：

```
from celery.signals import after_task_publish

@after_task_publish.connect
def task_sent_handler(sender=None, body=None, **kwargs):
    print 'after_task_publish: task_id: {body[id]}; sender: {sender}'.format(
        body=body, sender=sender)
```

现在执行 add 任务后会打印类似如下的信息：

```
after_task_publish: task_id: 89c089d7-bcb5-4d7f-8281-f159bb8c1afa; sender: proj.tasks.
add
```

信号可以帮助我们了解任务执行情况，分析任务运行的瓶颈。

Worker 管理

之前通过如下命令启动 Worker：

```
celery -A proj worker -l info
```

这种方式不怎么好管理，因为每次关闭时都需要使用 Ctrl + C。Celery 提供 multi 这个子命令来管理。我们先使用 Daemon 的方式启动 Worker 进程：

```
> celery multi start web -A proj -l info --pidfile=/tmp/celery_%n.pid --logfile=/tmp/
celery_%n.log
```

web 是对项目启动的标识，之后的处理都使用这个标识来管理。--pidfile 和 --logfile 使用了 %n，这是对节点的格式化用法，支持的格式化参数如表 9.5 所示。

表 9.5 格式化参数及其含义

参 数	含 义
%n	只包含主机名
%h	包含域名的主机名
%d	只包含域名
%i	Prefork 类型的进程索引，如果是主进程，则为 0
%I	带分隔符的 Prefork 类型的进程索引。假设主进程是 worker1，那么进程池的第一个进程则为 worker1-1

现在可以使用如下命令管理：

```
> celery multi show web # 查看web启动时的命令
/home/ubuntu/web_develop/venv/bin/python -m celery worker --detach -n web@WEB --pidfile
=web.pid --logfile=web.log --executable=/home/ubuntu/web_develop/venv/bin/python
> celery multi names web # 获取web的节点名字
web@WEB
> celery multi stop web # 停止web进程
celery multi v3.1.18 (Cipater)
> web@WEB: DOWN
> celery multi restart web # 重新启动web
celery multi v3.1.18 (Cipater)
> web@WEB: DOWN
> Restarting node web@WEB: OK
> celery multi kill web # 杀掉web进程
celery multi v3.1.18 (Cipater)
Killing node web@WEB (23675)
```

监控和管理 Celery

Celery 提供了一些方便监控和管理的命令，常用的命令有如下 4 个。

1. shell: celery 也提供了一个 Python 交互环境，内置了 Celery 应用实例和全部已注册的任务，支持默认的 Python 解释器、IPython 和 BPython。

```
> celery shell -A proj
In [1]: celery
Out[1]: <Celery proj:0x7fad4aef8290>
In [2]: add.delay(1, 2) # 可以直接使用而不需要import
```

2. result: 通过 task_id 在命令行获得执行结果。

```
> celery -A proj result eba7fc0b-3c16-4337-990c-d53a4c176a49
3
```

3. inspect active: 列出当前正在执行的任务。

```
> celery -A proj inspect active
```

4. inspect stats: 列出 Worker 的统计数据，常用来查看配置是否正确以及系统的使用情况。

```
> celery -A proj inspect stats
```

Celery 还可以撤销任务：

```

In : rs = add.delay(1, 2)
In : rs.revoke() # 只是撤销, 如果任务已经在执行则撤销无效
In : rs.task_id
Out: 'd24a83e8-e985-48ef-adab-e779441e5557'
In : app.control.revoke('d24a83e8-e985-48ef-adab-e779441e5557') # 通过task_id撤销
In : app.control.revoke('d24a83e8-e985-48ef-adab-e779441e5557', terminate=True) # 撤销
    正在执行的任务, 默认使用TERM信号
In : app.control.revoke('d24a83e8-e985-48ef-adab-e779441e5557', terminate=True, signal
    ='SIGKILL') # 撤销正在执行的任务, 使用KILL信号
In : app.control.revoke([
.....: 'd9078da5-9915-40a0-bfa1-392c7bde42ed'
.....: 'd24a83e8-e985-48ef-adab-e779441e5557'
.....: '40fa44d1-d1b4-4e3d-9e9d-2cd8f6cfd676']) # 可以同时撤销多个任务

```

Worker 进程在内存中保存了撤销的任务, 一重启就会丢失, 如果需要这个历史记录持久化, 可以添加`--statedb` 参数启动 Worker:

```
> celery -A proj worker -l info --statedb=/tmp/worker.state
```

Celery 官方推荐实时的 Web 监控工具 Flower (<https://github.com/mher/flower>), 它实现了如下特性:

- 可以看到任务历史、任务具体的参数、开始时间等。
- 提供图表和统计数据。
- 实现全面的远程控制功能, 包括但不限于撤销/终止任务、关闭和重启 Worker 进程、查看正在运行的任务等。
- 提供一个 HTTP API, 方便集成到运维系统中。

同时, Celery 也自带了一个事件监控工具显示任务历史等信息, 可以用来检查任务和跟踪错误:

```
> celery -A projb events
```

需要注意的是, 要把设置 `CELERY_SEND_TASK_SENT_EVENT` 为 `True` 才可以获取事件。

事件监控工具默认每秒都会刷新终端, 从消息代理以 `celeryev` 开头的队列中找到执行任务的历史记录, 可以使用键盘按键查看任务信息、错误堆栈、执行结果和撤销任务等。

子任务

我们可以把任务通过签名的方法传给其他任务, 成为一个子任务:

```
In : from celery import signature
```

```
In : task = signature('tasks.add', args=(2, 2), countdown=10)
In : task
Out: tasks.add(2, 2) # 通过签名生成了任务
In : task.apply_async()
Out: <AsyncResult: 05fbccf3-092f-405f-8e16-28de911804f2>
```

还可以通过如下方式创建子任务：

```
In : from proj.tasks import add
In : task = add.subtask((2, 2), countdown=10) # 可以使用
In : task.apply_async()
Out: <AsyncResult: a402fe1d-ed11-48f8-b163-88d7f65ab0e3>
```

`add.subtask` 同样可以使用快捷方式 `add.s(2, 2, countdown=10)`。

子任务实现偏函数（Partial）的方式非常有用，这种方式可以让任务在传递过程中才传入参数。

```
In : partial = add.s(2)
In : partial.apply_async((4,))
Out: <AsyncResult: b3aaa2ce-eddb-44eb-80fb-48ce5e36da00>
```

子任务支持如下 5 种原语来实现工作流。原语表示由若干条指令组成的，用于完成一定功能的过程。

1. `chain`：调用链。前面的执行结果作为参数传给后面的任务，直到全部完成。

```
In : from celery import chain
In : res = chain(add.s(2, 2), add.s(4), add.s(8))()
In : res.get()
```

`chain` 也可以使用管道（`|`）：

```
In : (add.s(2, 2) | add.s(4) | add.s(8))().get()
Out: 16
```

2. `group`：一次创建多个（一组）任务。

```
In : from celery import group
In : res = group(add.s(i, i) for i in range(10))()
In : res.get()
Out: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. `chord`：等待任务全部完成时添加一个回调任务。

```
In : res = chord((add.s(i, i) for i in range(10)), add.s(['a']))()
In : res.get() # 执行完前面的循环，把结果拼成一个列表之后，再对这个列表添加'a'
Out: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, u'a']
```

4. `map/starmap`: 每个参数都作为任务的参数执行一遍, `map` 的参数只有一个, `starmap` 支持的参数有多个。

```
In : ~add.starmap(zip(range(10), range(10)))
Out: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

它相当于:

```
@app.task
def temp():
    return [add(i, i) for i in range(10)]
```

5. `chunks`: 将任务分块。

```
In : res = add.chunks(zip(range(50), range(50)), 10)()
In : res.get()
Out:
[[0, 2, 4, 6, 8, 10, 12, 14, 16, 18],
 [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
 [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
 [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
 [80, 82, 84, 86, 88, 90, 92, 94, 96, 98]]
```

进阶篇: Celery 最佳实践

使用自动扩展

多进程和 `Gevent` 模式的 `Worker` 支持自动扩展, 通过 `--autoscale` 参数就可以实现:

```
> celery -A proj worker -l info --autoscale=6,3
```

`--autoscale` 参数接受 2 个数字。在上面的句子中, “`--autoscale=6,3`” 表示进程池平时保持 3 个进程, 最大并发进程数可以达到 6 个。这里的取值和你的服务器 CPU 个数、任务闲忙程度、可用内存等指标有关。笔者个人的习惯是将最大并发数设置为 CPU 个数的 2 倍。理论上 `Worker` 进程数与 CPU 个数接近即可, 但是也不尽然, 它还与你的任务类型有关。比如笔者曾经遇到过的一个产品线的 `Worker` 数就比 CPU 数高很多, 原因是每天上千万的任务中有大量的小任务, 也有耗时比较久的任务, 而且添加的任务会有瞬时高峰, 比如每小时的整点、偶数小时的整点等。这很容易造成多个消费者在处理耗时的任务, 让整体的及时性受到影响, 而使用自动扩展可以比较好地解决这个问题。

善用远程 Debug

Celery 支持远程使用 `pdb` 调试任务, 非常方便。

先添加一个任务：

```
from celery.contrib import rdb

@app.task
def sub(x, y):
    result = x - y
    rdb.set_trace() # 设置断点
    return result
```

重启 Worker 进程之后，使用 IPython 发布任务：

```
In : from proj.tasks import sub
In : sub(2, 1)
Remote Debugger:6899: Please telnet into 127.0.0.1 6899.
```

Type `exit` in session to continue.

Remote Debugger:6899: Waiting for client...

这个时候我们开启一个新的终端，使用 telnet 连接到 6899 端口：

```
> telnet localhost 6899
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
> /home/ubuntu/web_develop/chapter9/section4/proj/tasks.py(27)sub()
-> return result
(Pdb) result
1
(Pdb) x
2
(Pdb) continue
Connection closed by foreign host.
```

通过这样的方式，就不用在本地搭建 Celery 环境模拟任务失败的环境了。

合理安排任务周期

如果项目中有很多调度模式的任务，就要合理安排这些定时任务的执行时间。很多人会偷懒，随便选择一些整点和凌晨期间作为任务执行时间。但是需要注意如下问题：

1. 需要和系统管理员和数据库管理员确认，确保你选择的时间段里面没有一些不合时宜的时间点。因为通常在凌晨，尤其是过了 12 点之后也是大量定时任务运行的时间，比如数据备份、生成前一天的数据报表这样的任务，如果选在凌晨执行你的任

务，既可能让你的任务失败，也可能由于对数据库的访问压力让你和别人的任务花费更多的时间。

2. 选择非整点的任务执行周期。根据任务特性，可以把任务执行时间打散，尽量减少每小时里前 45 分钟 Celery 集群非常忙碌，后 15 分钟集群大部分空闲的情况。
3. 合理安排数据库、文件系统写任务。笔者的方式是把占用时间的数据库和文件系统写操作等任务安排在访问低峰期，并把写任务拆分后在分散的时间执行。

合理使用队列和优先级

不要把任务都放在默认队列。合理安排任务的优先级和队列，让应该及时完成的任务不会因为某些原因被阻塞而影响用户体验。其次是合理使用 `apply_async` 方法临时性地切换队列和优先级，提高整体的响应。

如果能明确知晓某些任务耗时很长，某些任务耗时很短，至少可以按照任务花费的时间来把任务安排到不同的队列。

保证业务逻辑的事务性

Celery 虽然提供错误重试机制，但是并没有提供任务的事务性。如果任务一部分执行成功之后失败，执行成功的那部分是没有回滚方法的，所以一开始就要在实现逻辑的时候对重试机制有明确的理解。

关闭你不想要的功能

如果你对任务的执行结果没有兴趣，可以关闭它：

```
@app.task(ignore_result=True)
def mytask(...):
    doit()
```

如果项目不需要限速，可以设置 “`CELERY_DISABLE_RATE_LIMITS = True`” 直接全局地把这个特性关闭。

使用阅后即焚模式

在使用队列的时候，默认使用持久化方式以确保任务被执行，如果你的任务不需要持久化，可以使用阅后即焚（`transient`）模式：

```
from kombu import Queue
Queue('transient', routing_key='transient',
     delivery_mode=1)
```

善用 Prefetch 模式

Worker 进程默认每次从消息代理获取 CELERYD_PREFETCH_MULTIPLIER 设置的任务数，假如你的任务都很细小，可以修改 CELERYD_PREFETCH_MULTIPLIER 的值，每次获取更多的任务数。这个值也是根据任务的执行情况决定的。

善用工作流

如果一个任务有调用链，下一步任务需要等待上一步的结果，就不应该使用同步子任务。举一个抓取某电商网站的爬虫例子，它一般分为以下几步：

1. 获取要抓取的页面。
2. 抓取对应页面。
3. 解析页面数据。
4. 把需要的数据存储到数据库和缓存中。

如果设计成下面的方式就是错误的：

```
@app.task
def page_crawler():
    url = get_url.delay().get()
    page = fetch_page.delay(url).get()
    info = parse_page.delay(page).get()
    store_page_info.delay(info)

@app.task
def get_url():
    return PageInfo.objects.filter_by(need_update=True).first()

@app.task
def fetch_page(url):
    return requests.get(url)

@app.task
def parse_page(page):
    return myparser.parse_document(page)
```



```
@app.task
def store_page_info(info):
    return PageInfo.objects.create(info)
```

应该使用如下方式：

```
def page_crawler():
    # get_url -> fetch_page -> parse_page -> store_page
    chain = get_url.s() | fetch_page.s() | parse_page.s() | store_page_info.s()
    chain()
```

控制任务的粒度不要太细。举个例子，假设现在每小时都要通过某网站的开放平台获取对应的百万级条目的数据。如果为每个条目都创建一个任务显然是不合理的，应该更多地利用开放平台接口的限制。比如接口可以批量操作，一次最多取 30 个条目的数据，那么可以考虑把单个任务定为处理 30 个条目的数据。

在生成任务的时候应该充分利用 `group/chain/chunks` 这些原语。

第 10 章

服务化

本章将介绍如下内容：

- 为什么需要服务化。
- 使用 Thrift 对文件托管服务改造。
- 介绍豆瓣服务化实践——PIDL 的起因、基本原理和基本架构。

为什么需要服务化

当网站流量很小的时候，通常全部功能都部署在一起。等访问量增加到一定程度，把单一应用拆分成几个应用就好了。但随着业务的发展，应用越来越多，越来越复杂，开发团队规模也越来越大，会出现如下几个棘手的问题。

1. 代码重复率高。例如 A 产品有一个功能，用户反馈很不错。B 产品也想加进来，这时候 B 产品的工程师会有两种选择：
 - 为了快速验证产品，拷贝了 A 的代码再改一改就上线了。拷贝别人的代码，尤其没有非常清晰了解其代码意图时，未来出问题的概率非常高，维护成本也很高。
 - 重新实现一遍。需要花费的时间多，还有一个隐患是未来几乎不可能融合两个产品，一般需要重新对 A 和 B 梳理后重新实现。

笔者曾经见过某个功能模块，在不同产品线和同一产品线不同页面有十几个版本。每次需求改动都要对这些模块进行同步修改，到最后不得不花费很大的精力使用统一的实现替换它们。

2. 运维成本高。业务没有拆分，哪怕是一个功能模块的小改动也需要对整个业务重新部署上线。
3. 可靠性很差。网站页面通常集合了多个产品线的功能，应用之间的交互和依赖不可避免，其中一个产品线出了问题就可能造成全站都不可用。
4. 团队协作差。公司越大，推动一些公共技术架构越困难，团队越分越散，最后的结果是很难协调一起开发，大家各自为营。

解决办法如下：

- 业务拆分。把巨无霸业务分拆成一系列小的业务。
- 业务隔离。拆分的业务可以独立地部署、扩容、升级等。
- 业务解耦。通过服务化、订阅/发布机制等手段让应用调用关系解耦。
- 明确业务划分。明确服务调用方和开发维护方职责，通过服务化作为技术契约的约束等方式，让团队间的沟通和共享工作更顺畅。
- 减少重复开发。通过熟悉产品代码的人把关，减少相似功能的立项，再加上一些代码检查工具，减少重复代码。

服务化的划分常常有两种维度：

- 根据业务。比如，豆瓣电影是一个产品线。把豆瓣电影独立成一个服务，被其他产品线使用。整个豆瓣主站再也不会由于一个产品线的问题而让全站不可用。
- 根据功能。豆瓣的读书、音乐、电影等产品线都有长评这个功能。把长评功能独立出来，被这些产品线使用。服务化之后，那些只需要长评功能的产品线不再需要依赖庞大复杂的豆瓣电影服务，直接调用长评服务即可。

RPC 框架

RPC（Remote Procedure Call，远程过程调用）是一个计算机通信协议，此协议允许进程间通信。RPC 框架屏蔽了底层的传输方式（TCP/UDP）、序列化和反序列化（XML/JSON/二进制）等内容，使用框架只需要知道被调用者的地址和接口就可以了，无须额外地为这些底层内容编程。

目前主流的 RPC 框架有如下几种。

- Thrift：Facebook 开源的跨语言框架，在豆瓣内应用广泛。
- Avro：Hadoop 的子项目。
- gRPC：Google 基于 HTTP/2 和 Protobuf 的通用框架。

这个时候可能你会产生疑问：使用 RESTful API 也可以实现和远程服务的通信，为什么要选择 RPC 呢？我们从如下两个方面对比：

- 资源粒度。RPC 就像本地调用方法，RESTful API 每一次添加接口都可能需要额外地组织开放接口的数据，这相当于在应用视图中再写了一次方法调用。而且它还要维护开放接口的资源粒度、权限等。
- 流量消耗。RESTful API 在应用层使用 HTTP 协议，哪怕使用轻型、高效、传输效率更高的 JSON 也会消耗较大的流量，而 RPC 的协议一般使用二进制编码，大大降低了数据的大小，减少流量消耗。



对接异构第三方服务时，通常选用 HTTP/RESTful 等公有协议；对于内部的服务调用，应该选择性能更高的二进制私有协议。

服务化带来的问题

服务化之后，客户端和服务端的交互一般变成了远程网络通信，序列化和反序列化、网络传输等工作势必增加性能损耗，所以客户端在请求中尽量不要包含不需要的服务调用，同时也需要对请求合理地合并来减少网络请求。

服务调用链越复杂，出现问题时就越不好定位，这需要一个分布式跟踪系统帮助我们准确地判断问题。它追踪每个请求的完整调用链路，收集调用链路上每个服务的性能数据。豆瓣服务化的实时分布式跟踪系统 Shuai 是基于 Twitter 开源的 Zipkin (<http://bit.ly/2bq57b0>) 和 Google 的 Dapper 论文等实现的纯 Python 系统。Zipkin 和 Shuai 都可以把搜集的数据用瀑布的方式在 Web 页面上直观地展示出来，从 Shuai 上能很方便地看到调用链路上每一步做的事情、服务接口信息、使用的代码的具体位置、执行代码所花费的时间等。Shuai 还能生成各种指标图，支持高级查询等。

微服务架构

微服务架构是一种架构模式，它提倡将应用分解为非常小的、原子的微服务，尤其现在云计算、Docker 容器、移动互联网等技术快速发展，并被大量应用，也让微服务这种架构风格越来越受关注，为人使用。

微服务架构的好处如下：

- 每个微服务都专注实现一个特定功能，界限明确。比如上面说的长评功能。
- 微服务可以独立地开发、持续集成和部署。新功能上线周期非常短。

- 可以用不同的开发语言来开发。
- 微服务易于理解，由于功能相对单一，代码量很少，开发人员容易修改和维护，新的团队成员也可以很快融入到开发中。
- 微服务相对没有历史包袱，更容易采用最新的技术（如框架、编程语言、编程实践等）。

但是它也有如下的缺点：

- 显著增加运维成本，因为需要做多得多的配置、部署、扩展和监控的工作。
- 服务数量级别很大，管理整个系统很麻烦。
- 微服务依赖可能影响开发、测试和上线的效率，尤其是集成测试不可避免大量服务的依赖，这需要花费更多的精力来保证接口能正常调用。举个例子，假设有 A、B、C 三个服务，A 依赖 B，B 依赖 C。原来单个服务中直接整合变化，重新部署就可以了，在微服务中需要考虑相关改变对不同服务的影响，还要多考虑修改和上线的顺序。
- 选择不同的开发语言时，如果对性能也有较高的要求，会造成整体链路上的复杂度就呈指数级上升。通常应该标准化一个主要的技术栈（解决 80% 的主要问题），只在特殊情况下允许少量的差异化生产（20% 的特殊情况）。

微服务更适合规模较大的公司或者研发团队。

使用 Thrift

Apache Thrift 最初是 Facebook 实现的一种支持多种编程语言、高效的远程服务调用框架，2008 年进入 Apache 开源项目。它采用中间语言（IDL，接口描述语言）定义 RPC 的接口和数据类型，通过一个编译器生成不同语言的代码（支持 C++、Java、Python、Ruby 等多种语言），其数据传输采用二进制格式，相对 XML 和 JSON 而言体积更小，对于高并发、大数据量和多语言的环境更有优势。

我们先安装它：

```
> wget http://mirrors.cnnic.cn/apache/thrift/0.9.3/thrift-0.9.3.tar.gz
> tar xzf thrift-0.9.3.tar.gz
> cd thrift-0.9.3
# 由于安装RabbitMQ的时候安装了Erlang，可以禁用Erlang
> ./configure --without-erlang
> make && sudo make install
> cd lib/py
> sudo make install
```

安装 Thrift 的 Python 库的时候提示了包的安装路径，在 server.py 一开始要指定这个包路径：

```
import sys

sys.path.append('gen-py')
sys.path.append('/usr/lib/python2.7/site-packages')
```

定义 IDL 文件

我们将把文件托管项目服务化。首先定义 thrift 文件（pastefile.thrift）：

```
struct PasteFile {
    1: required i32 id,
    2: required string filename,
    3: required string filehash,
    4: required string filemd5,
    5: required string uploadtime,
    6: required string mimetype,
    7: required i64 size,
    8: required string url_s,
    9: required string url_i,
    10: required list<i32> image_size,
    11: required string url_d,
    12: required string url_p,
    13: required string size_humanize,
    14: required string type,
    15: required string quoteurl,
}

struct CreatePasteFileRequest {
    1: required string filehash,
    2: required string filename,
    3: required string mimetype,
    4: optional i32 width,
    5: optional i32 height,
}

exception ImageNotSupported {
    1: string message
}

exception UploadImageError {
    1: string message
}

exception NotFound {
```

```
    1: i32 code
}

exception ServiceUnavailable {
    1: string message
}

service PasteFileService {
    PasteFile get(1:i32 pid)
        throws(
            1: ServiceUnavailable service_error,
            2: NotFound not_found
        ),
    list<string> get_file_info(1:string filename, 2:string mimetype)
    PasteFile create(1: CreatePasteFileRequest request)
        throws(
            1: ServiceUnavailable service_error,
            2: ImageNotSupported error,
            3: UploadImageError image_error
        ),
}
```

解析一下.thrift 文件的语法：

1. struct 关键字表示 Thrift 的结构体，概念上类似于一个 C 结构体，它将相关属性组合在一起。
2. Thrift 要求预先定义好字段和返回值类型，i32、i64、string 等都是 Thrift 内置的类型，当然也可以自定义类型。
3. list 表示有序列表。除此之外还支持 map（Python 中的字典）和 set（无序不重复元素集）。
4. exception 关键字表示 Thrift 的异常。
5. service 是 Thrift 的服务接口（类似于 Python 的方法）。其中包含三个接口：get、get_file_info 与 create，每个接口参数不同，但是需要定义参数的类型和顺序。每行定义的第一个字段表示接口返回的类型，比如 PasteFile get(1:i32 pid) 表示执行 get 接口返回一个 PasteFile 类型的对象。需要注意，不一定返回的对象都是定义的结构体，也可以是内置的类型，比如 get_file_info 接口返回的就是一个字符串的列表。
6. throws 块内列出了可能抛出的异常。

生成 Thrift 代码：

```
> thrift -r --gen py pastefile.thrift
```

生成的代码目录结构如下：

```
> tree gen-py
gen-py
├── __init__.py
└── pastefile
    ├── constants.py
    ├── __init__.py
    ├── PasteFileService.py # 存放生成的服务代码
    ├── PasteFileService-remote # 自动生成的client实例，可以在命令行请求服务接口
    └── ttypes.py # 存放生成的结构体、异常的代码
```

服务端实现

先引入 Thrift 相关的模块：

```
from thrift.transport import TTransport, TSocket
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer
```

```
from pastefile import PasteFileService
from pastefile.ttypes import PasteFile, UploadImageError, NotFound
```

基于 PasteFile 实现了 RealPasteFile，需要重载两个方法。第一个是 get_url 方法，原来的用法是：

```
from flask import request
```

```
def get_url(self, subtype, is_symlink=False):
    hash_or_link = self.symlink if is_symlink else self.filehash
    return 'http://{host}/{subtype}/{hash_or_link}'.format(
        subtype=subtype, host=request.host, hash_or_link=hash_or_link)
```

由于 request 来自 Flask，服务化之后不能使用 request.host，需要在响应之前再填充 host 这个参数，修改之后是这个样子：

```
def get_url(self, subtype, is_symlink=False):
    hash_or_link = self.symlink if is_symlink else self.filehash
    return 'http://%s/{subtype}/{hash_or_link}'.format(
        subtype=subtype, hash_or_link=hash_or_link)
```

也就是预先留了一个 %s 占位，等待 API 返回之前再拼进去。

另一个要修改的是 `create_by_upload_file`，之前参数 `uploaded_file` 是一个上传的文件对象，如果做了服务化则需要把这个对象通过 `client` 传给 `server`，再由 `server` 保存，这相当于增加了复杂度和网络延迟，所以直接保存文件，在 `create_by_upload_file` 中只判断重复：

```
@classmethod
def create_by_upload_file(cls, uploaded_file):
    rst = uploaded_file
    with open(rst.path) as f:
        filemd5 = get_file_md5(f)
        uploaded_file = cls.get_by_md5(filemd5)
        if uploaded_file:
            os.remove(rst.path)
            return uploaded_file
    filestat = os.stat(rst.path)
    rst.size = filestat.st_size
    rst.filemd5 = filemd5
return rst
```

再定义服务处理的类，这个类其实就是接口的封装：

```
class PasteFileHandler(object):
    # 这一步比较绕，使用Flask保存文件在app.py中执行，没有必要传输到服务端再保存，需要预先生成文件路径
    def get_file_info(self, filename, mimetype):
        rst = PasteFileModel(filename, mimetype, 0)
        return rst.filehash, rst.path

    # 方法的参数类型已经在pastefile.thrift中定义了，request是一个CreatePasteFileRequest实例
    def create(self, request):
        width = request.width
        height = request.height
        filehash = request.filehash
        filename = request.filename
        mimetype = request.mimetype

        uploaded_file = PasteFileModel.get_path(filehash)
        uploaded_file.filename = filename
        uploaded_file.mimetype = mimetype
        try:
            if width and height:
                paste_file = RealPasteFile.rsize(uploaded_file, width, height)
            else:
                paste_file, _ = RealPasteFile.create_by_upload_file(
                    uploaded_file, filehash)
        except:
            raise UploadImageError()
```

```

        db.session.add(paste_file)
        db.session.commit()
        return self.convert_type(paste_file)

    def get(self, pid):
        paste_file = PasteFileModel.query.filter_by(id=pid).first()
        if not paste_file:
            raise NotFound() # 如果不使用预先定义的异常类, 抛出的异常都是
                             # TApplicationException
        return self.convert_type(paste_file)

    @classmethod
    def convert_type(cls, paste_file):
        '''将模型转化为Thrift结构体的类型'''
        new_paste_file = PasteFile()
        for attr in ('id', 'filehash', 'filename', 'filemd5', 'uploadtime',
                    'mimetype', 'symlink', 'size', 'quoteurl', 'size', 'type',
                    'url_d', 'url_i', 'url_s', 'url_p'):
            val = getattr(paste_file, attr)
            if isinstance(val, unicode):
                # 因为需要传输字符串, 所以对unicode要编码
                val = val.encode('utf-8')
            # Thrift不支持Python的时间格式, 需要转换一下, 在客户端再转换回来
            if isinstance(val, datetime):
                val = str(val)
            setattr(new_paste_file, attr, val)
        return new_paste_file

```

启动服务的代码如下:

```

import logging
logging.basicConfig() # 这一步很重要, 可以收到Thrift发出来的异常日志
handler = PasteFileHandler()
# Processor用来从连接中读取数据, 将处理授权给handler (自己实现), 最后将结果写到
# 连接上
processor = PasteFileService.Processor(handler)
# 服务端使用8200端口, transport是网络读写抽象层, 为到来的连接创建传输对象
transport = TSocket.TServerSocket(port=8200)
tfactory = TTransport.TBufferedTransportFactory()
pfactory = TBinaryProtocol.TBinaryProtocolFactory()

server = TServer.TThreadPoolServer(
    processor, transport, tfactory, pfactory)
print 'Starting the server...'
server.serve()

```

客户端实现

client.py 中同样先引入 Thrift 相关的定义：

```
from thrift.transport import TTransport, TSocket
from thrift.protocol import TBinaryProtocol

from pastefile import PasteFileService
from pastefile.ttypes import (
    PasteFile, CreatePasteFileRequest, UploadImageError,
    NotFound)
```

为了让客户端连接发生在服务器启动之后，而且能重用连接，我们使用了 LocalProxy 包装 client：

```
from werkzeug.local import LocalProxy

def get_client():
    # 同样使用8200端口，使用阻塞式I/O进行传输，是最常见的模式
    transport = TSocket.TSocket('localhost', 8200)
    transport = TTransport.TBufferedTransport(transport)
    # 封装协议，使用二进制编码格式进行数据传输
    protocol = TBinaryProtocol.TBinaryProtocol(transport)
    client = PasteFileService.Client(protocol)
    transport.open() # 打开连接
    return client

client = LocalProxy(get_client)
```

这样就可以在下面的逻辑中直接使用 client 了。

由于上传逻辑出现在两个视图中，所以抽象一个连接函数，让代码复用：

```
def create(uploaded_file, width=None, height=None):
    filename = uploaded_file.filename.encode('utf-8')
    mimetype = uploaded_file.mimetype.encode('utf-8')
    filehash, path = client.get_file_info(filename, mimetype)

    create_request = CreatePasteFileRequest()

    create_request.filename = filename
    create_request.mimetype = mimetype
    create_request.filehash = filehash

    # 接收上传文件，直接保存，没有必要传输到服务端再去保存
    uploaded_file.save(path)
```

```

if width is not None and height is not None:
    create_request.width = width
    create_request.height = height
try:
    pastefile = client.create(create_request)
except UploadImageError: # 异常是在PasteFileHandler的create方法中预先定义的
    return {'r': 1, 'error': 'upload fail'}

print isinstance(pastefile, PasteFile) # 只是验证

try: # 事实上没有必要重新get一次, 因为create方法已经返回了PasteFile实例, 这里
    只是演示
    paste_file = client.get(pastefile.id)
except NotFound:
    return {'r': 1, 'error': 'not found'}

return {'r': 0, 'paste_file': paste_file}

```

在 app.py 里面引用 create 方法, 并且应用于视图:

```
from client import create
```

```

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        uploaded_file = request.files['file']
        w = request.form.get('w', None)
        h = request.form.get('h', None)
        if not uploaded_file:
            return abort(400)

        # 使用Thrift客户端代码请求服务端之后获得创建的文件对象
        rs = create(uploaded_file, width=w, height=h)
        if rs['r']:
            return rs['error']

        paste_file = rs['paste_file']

    return jsonify({
        'url_d': paste_file.url_d % request.host, # 由于之前get_url的值中的主机
        名使用了%s占位, 这里填充进去
        'url_i': paste_file.url_i % request.host,
        'url_s': paste_file.url_s % request.host,
        'url_p': paste_file.url_p % request.host,
        'filename': paste_file.filename,
        'size': humanize_bytes(paste_file.size),
    })

```

```

        'uploadtime': paste_file.uploadtime,
        'type': paste_file.type,
        'quoteurl': paste_file.quoteurl.replace('%25s', request.host) # quoteurl已
                               经是url编码后的结果, 需要使用替换的方式
    })
    return render_template('index.html', **locals())

```

这样就实现了服务化: app.py 专注于视图逻辑; client.py 专注于请求服务; server.py 处理客户端发来的接口请求。实际生产环境中还可以考虑 gRPC 和 Nameko (<http://bit.ly/1W3N8qK>), 它们可作为实现 RPC 框架的参考。

如果你所在的生产环境只使用 Python 一种语言, 不喜欢 Thrift 生成的不符合 Python 编码规范的代码, 而且对 Python 实现的 RPC 情有独钟, 可以选择饿了么开源的纯 Python 实现的 thriftpy (<https://github.com/eleme/thriftpy>)。

我们先安装它:

```
> pip install thriftpy cython
```

安装 cython 是因为 thriftpy 的二进制协议实现可以使用 cython 加速。接下来实现一个 add 功能的服务。首先看 Thrift 定义 (calc.thrift):

```

service CalcService {
    i64 add(1:i64 a, 2:i64 b),
}

```

看一下服务端的实现 (server_with_thriftpy.py):

```

import os
import logging

import thriftpy
from thriftpy.rpc import make_server
from thriftpy.protocol import TBinaryProtocolFactory
from thriftpy.transport import TBufferedTransportFactory

HERE = os.path.abspath(os.path.dirname(__file__))
logging.basicConfig()

calc_thrift = thriftpy.load(
    os.path.join(HERE, 'calc.thrift'),
    module_name='calc_thrift')

class Dispatcher(object):
    def add(self, a, b):

```

```
        return a + b

server = make_server(calc_thrift.CalcService,
                     Dispatcher(),
                     '127.0.0.1', 8300,
                     proto_factory=TBinaryProtocolFactory(),
                     trans_factory=TBufferedTransportFactory())
print 'serving...'
server.serve()
```

启动它：

```
> python chapter10/section2/server_with_thrift.py
```

再看一下客户端的实现（client_with_thrift.py）：

```
import os

import thriftpy
from thriftpy.rpc import client_context
from thriftpy.protocol import TBinaryProtocolFactory
from thriftpy.transport import TBufferedTransportFactory

HERE = os.path.abspath(os.path.dirname(__file__))

calc_thrift = thriftpy.load(
    os.path.join(HERE, 'calc.thrift'),
    module_name='calc_thrift')

with client_context(calc_thrift.CalcService,
                   '127.0.0.1', 8300,
                   proto_factory=TBinaryProtocolFactory(),
                   trans_factory=TBufferedTransportFactory(),
                   timeout=None) as calc:
    rs = calc.add(1, 2)
    print 'Result is: {}'.format(rs)
```

现在执行这个客户端程序，就能获得两数相加的结果了：

```
> python client_with_thrift.py
Result is: 3
```

PIDL——豆瓣的服务化实践

Shire 是豆瓣主站代码仓库，包含了早期的项目代码、各产品线公用的代码、遗留代码等。虽然大部分的产品线已经拆分出去使用 DAE 来服务，但是还是通过软链接的方式被包含进去。一开始这种代码组织方式还是可以接受的，但随着项目逐渐变大，以下问题越来越严重：

- 大部分产品线都依赖 Shire，改动产品线 A 的代码可能会影响到产品线 B，甚至造成全站出错。
- 任何改动都需要 Shire 的人工上线，上线很慢，故障修复不及时。
- 任何改动的持续集成时间都很长。

PIDL 是豆瓣解决这些问题的方案，严格的说它不是一种语言，只是一个 Python 模块/包的接口隔离层。它把服务全部隐藏起来，只通过 PIDL 文件暴露那些需要被外部使用的接口。它的应用实现了如下目的：

1. 代码解耦。产品线的拆分更利于团队协作，大家再也不用往 Shire 上提 PR 了。
2. 故障隔离。产品线的故障不再会影响其他产品线。
3. 运维方便。产品线都使用 DAE 独立部署和监控，也方便扩容。
4. 服务化方案灵活。无论是对产品线，还是对产品线的某个功能实现服务化都很方便，工程师稍加培训就可以胜任服务化工作。
5. 对产品线开发方式的影响很小。虽然豆瓣在产品线某些功能上使用 Thrift，但是对整个产品线做类似的服务化意味着需要改动大量的逻辑，这是不可接受的。而使用 PIDL 对现有的业务逻辑几乎没有影响。
6. 对服务化高度可控。自主研发可以使用最简单的解决方案，保证了高度的可控性以及和豆瓣现有架构的兼容性，PIDL 也和 DAE 等基础设施关系紧密。
7. 优雅降级。当服务不可用或者失败，会返回预先定义的结果给客户端，不会让程序发生异常。可以容忍单点失败。

PIDL 和 Thrift 的区别在于：

- PIDL 文件直接使用，不需要对其编译。
- PIDL 文件不需要预先定义参数类型。
- PIDL 的结构体就是 Python 的类、对象、函数、常量等，更 Pythonic。
- PIDL 的接口可以有非常复杂的层级，写起来比 Thrift 的接口简洁得多。
- PIDL 基于 Pickle 的二进制协议，以牺牲语言无关性为代价，尽可能地减少对代码的修改。

PIDL 架构

假设现在有一个要服务化的产品 Story，它至少有如下 3 个文件。

1. PIDL 文件。对于 Story，通常叫作 story_pidl.py。它包含了 PIDL 设置、PIDL 接口声明和返回的默认值等：

```
import pidl # 文件中不能引用业务中的模块

__pidl_config__ = {
    'name': 'story',
    'implemented_by': 'story_service', # 指定后端逻辑入口模块
    'active_plugins': [
        'monitor', # PIDL具有很高的扩展性，默认已经注册了一些插件，这里可以列出来那些默认未激活而需要激活的插件
    ]
}

def get_stories_by_author_id(author_id): # 定义的接口可以是函数、类、对象、变量等
    return [] # Fallback模式，当服务不可用或者失败时返回给客户端的默认值

class Story(object):
    story_id = 0
    author_id = None
    category_id = 100
    creation_time = None
    def __init__(self, story_id, author_id, category_id, creation_time):
        pass

    def get(story_id):
        return ''

    @classmethod
    def get_stories_by_category_id(cls, category_id):
        return []

@pidl.retry(10) # retry和timeout是默认已注册的插件，一般通过装饰器来激活插件
@pidl.timeout(100)
def refresh_stories_indices():
    pass
```

2. PIDL 输出文件。一般叫作 story_export.py，客户端通过这个文件和服务端通信，它把 PIDL 封装好的对应接口都内嵌到这里面，比如客户端调用上面定义的

get_stories_by_author_id 函数就会是这样：

```
from story_export import get_stories_by_author_id
```

返回的是 PIDL 接口对象，这个对象带了那些需要的属性、方法等。

3. 后端逻辑入口文件。通常叫作 story_service.py，存放对应 story_pidl.py 文件中提供的接口的真实引用：

```
from story.models.bot import refresh_stories_indices
from story.models.story import Story, get_stories_by_author_id
```

要保证 story_service.py 和 story_pidl.py 的接口一一对应。

要确保 Fallback 模式下返回值的类型和正确返回的值的类型一致。设想本来返回值是一个列表，逻辑上对返回结果做 for 循环，如果在 Fallback 模式返回 None，就会在执行 for 循环的时候抛出 TypeError 异常。

story_export.py 的输出模式有两种。

1. 内嵌模式。常用于持续集成，减少服务化对测试的影响和复杂度。其实原始的模块/包都还在，只是通过 PIDL 把真实的模块/包和实际调用隔离开，也就是永远不能使用“from story_service import get_stories_by_author_id”的方式引用：

```
import pidl
import story_pidl

server = pidl.implement(story_pidl)
server.embed(globals())
```

2. RPC 模式。在生产环境使用，通常是远程调用。通过 PSGI（Pidl Server Gateway Interface，基于 WSGI）协议，用独立模式启动 PIDL 服务：

```
pidl-server -b "127.0.0.1:8300" story_pidl
```

选择如下的方式调用：

```
import pidl
import story_pidl

client = pidl.make_client(story_pidl, server='pidl://127.0.0.1:8300')
client.pidl_embed(globals())
```

RPC 模式的架构如图 10.1 所示。

二进制协议是高可用 RPC 框架必须支持的协议，PIDL 的二进制网络协议借鉴了 SPDY（<http://bit.ly/2b5LByy>）和 MessagePack-RPC（<http://bit.ly/2aSvSDi>）。

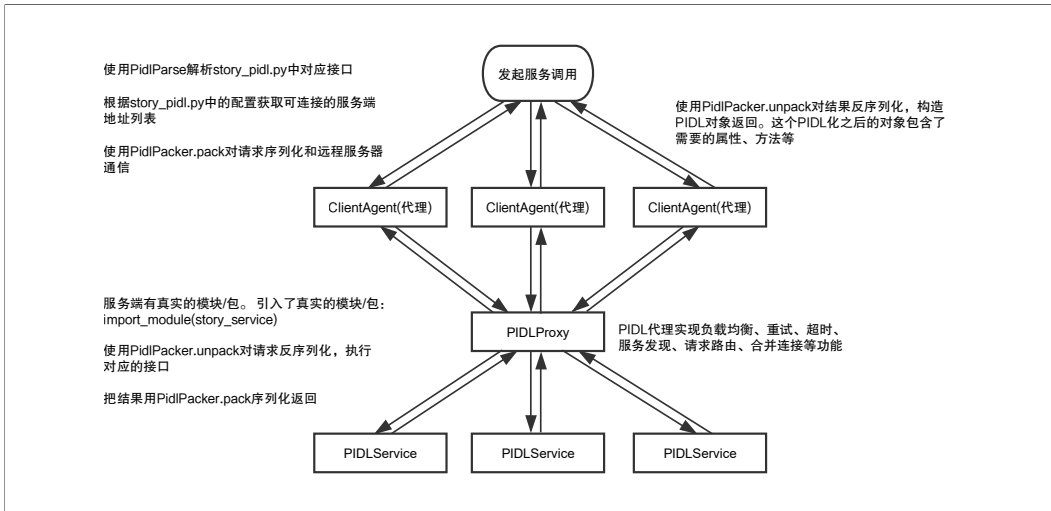


图 10.1 RPC 模式的架构

第 11 章

数据处理

数据分析和处理是 Web 开发一项很平常的工作，然而要做好却不容易，本章基于笔者的实际工作经验，从数据报表、日志分析等角度列出多个真实实例帮助读者了解这项工作。本章主要包含如下内容：

- 使用纯 Python 代码实现 MapReduce 功能。
- 配置 DPark 环境，深入了解 DPark，演示如何用 DPark 对业务日志进行 PV 和 UV 的分析。
- 通过发送带有样式和附件的邮件，创建包含带样式和 Sparkline 图表的 xlsx 文件，以及创建包含多工作表和图表的 xlsx 文件这三个有用的例子，展示笔者对数据报表的理解和运用。
- 基于数据报表中的数据，用 Pandas 进行分析和展示。

使用 MapReduce 做日志分析

使用 MapReduce

MapReduce 是谷歌提出的一个编程模型，它把对数据集的大规模操作拆分后分发给网络上的多个节点，每个节点会周期性地把完成的工作和状态的更新报告回来，以实现并行计算的目的。Map 表示映射，一个映射函数读取被分配的一小部分数据任务，计算之后输出中间的键值对的集合；Reduce 表示归纳，一个归纳函数收集具有相同中间键的值，合并这些值，形成一个较小的值的集合。

在开始使用 MapReduce 之前，先看一个使用 multiprocessing 实现并发计算的例子（parallel.py）：

```
import os
import time
import multiprocessing

def task(args):
    time.sleep(1)
    pid = os.getpid()
    return pid, args

start = time.time()
pool = multiprocessing.Pool(processes=4)
result = pool.map(task, range(10))
print result
print 'Cost: {}'.format(time.time() - start)
```

把 0~9 这 10 个数作为参数传给函数 task，每个任务都 sleep 1 秒，完成 10 个任务需要多久呢？执行一下看看：

```
> python ~/web_develop/chapter11/section1/parallel.py
[(8568, 0), (8569, 1), (8570, 2), (8571, 3), (8568, 4), (8569, 5), (8570, 6), (8571, 7),
 , (8568, 8), (8569, 9)]
Cost: 3.05770802498
```

虚拟机是双核 CPU，上例开启了 4 个进程，从执行的结果可以发现，执行任务的就是这 4 个进程，更好地利用了虚拟机的 CPU 资源；同时也可以看到使用 multiprocessing.Pool 的 map 方法实现了带有队列功能的并发。

有一个从某段日期的日志中获得符合条件的记录的需求，其实现步骤如下：

1. 解压缩日志。为了节省空间，早期的日志会压缩后再存储，每天压缩后的日志约 200 GB，由非常多的小文件组成，单是解压缩这些文件就需要很长时间。
2. 遍历每一行记录，记录的字段按\t 分隔，需要找到符合条件的记录。
3. 在符合条件的全部记录中，统计符合不同条件的记录数量。

这个需求使用单进程串行的方式运行，完成一次耗时超过 24 小时，且占用大量内存。笔者接手之后改进了一版：

1. 遍历过程中定期执行垃圾回收。
2. 修改为多进程的方式，启动与 CPU 核数（24 核）相同的进程，每个进程通过“hash(filename) % 24 == 0”的方式只执行对应的 1/24。每个进程把符合条件的记录

存入 SQLite 数据库。全部执行完毕后查询 SQLite 获得结果。

这种方法可以把时间缩短到 2 个多小时，比之前的版本效率提高了 10 倍。但还是有不合理的地方：有些日志文件相对较大，需要的时间要多一些，累积起来的结果就是有些进程先跑完了，但是还要等那些跑得慢的进程完成，等待的过程中先跑完的进程是闲置的，并没有被充分利用。

这个时候当然可以使用队列实现进程间通信，把日志文件放入队列，其他进程从队列中取数据执行，但这样做会增加代码的复杂度。我们依据 `pymotw` (<http://bit.ly/1WQsuvq>) 上的例子完成最后一版 (`map_reduce.py`)。

首先定义抽象的 `MapReduce` 类：

```
import collections
import itertools
import multiprocessing

class MapReduce(object):
    def __init__(self, map_func, reduce_func, num_workers=None):
        self.map_func = map_func # map函数
        self.reduce_func = reduce_func # reduce函数
        # num_workers为None会使用与CPU核数相同数量的进程
        self.pool = multiprocessing.Pool(num_workers)

    def partition(self, mapped_values):
        partitioned_data = collections.defaultdict(list)
        for key, value in mapped_values:
            partitioned_data[key].append(value)
        return partitioned_data.items()

    def __call__(self, inputs, chunksize=1):
        # inputs是一个需要处理的列表，chunksize表示每次给mapper的量，根据需求调整这个值
        # 第一次pool.map是为了把大任务分组
        map_responses = self.pool.map(
            self.map_func, inputs, chunksize=chunksize)
        # chain把mapper的结果链接为一个可迭代的对象
        partitioned_data = self.partition(itertools.chain(*map_responses))
        # 第二次pool.map是为了聚合结果实现reduce，map方法继续用来实现并行计算
        reduced_values = self.pool.map(self.reduce_func, partitioned_data)
        return reduced_values
```

看完这个类就可以明白，`Map` 和 `Reduce` 是借用 `map` 方法实现的。`MapReduce` 接收两个必选参数：`map` 函数和 `reduce` 函数。`map` 函数接收的参数是需要处理的数据，返回过滤后的结果：

```
import bz2
```

```
def mapper_match(one_file):  
    '''第一次的map函数，从每个文件里面获取符合的条目'''  
    output = []  
    for line in bz2.BZ2File(one_file).readlines():  
        l = line.rstrip().split()  
        if l[3] == 'web' and l[5] == '0':  
            output.append((l[4], 1))
```

mapper_match 函数的返回值是 “[(a, 1), (b, 1)]” 这样的第二个值为 1 的元组组成的列表。

reduce 函数用来对函数返回的结果进行归纳：

```
def reducer_match(item):  
    cookie, occurrences = item  
    return (cookie, sum(occurrences))
```

归纳听起来不好理解，举个例子，假设 mapper 返回了 “[(‘a’, 1), (‘b’, 1), (‘a’, 1)]”，reducer 函数对这三个元素遍历，对相同的第一个元素聚合，reducer_match 函数的返回值就是 “[(‘a’, 2), (‘b’, 1)]”。

我们还会基于第一次的结果进行第二次 MapReduce 操作，这次把符合条件的元素的数量作为键：

```
def mapper_count(item):  
    _, count = item  
    return [(count, 1)]
```

```
def reducer_count(item):  
    freq, occurrences = item  
    return (freq, sum(occurrences))
```

假设第一次 MapReduce 的返回值是 “[(‘a’, 2), (‘b’, 1), (‘c’, 2), (‘d’, 3)]”，第二次的结果就是 “[(1, 1), (2, 2), (3, 1)]”，表示出现 1 次的键（b）有 1 个，出现 2 次的键（a 和 c）有 2 个，出现 3 次的键（d）也只有 1 个。

完整的日志处理过程如下：

```
import glob  
import operator  
  
input_files = glob.glob(  
    '/home/ubuntu/web_develop/chapter11/section1/data/*.bz2')  
mapper = MapReduce(mapper_match, reducer_match)
```

```
cookie_feq = mapper(input_files)
print 'Result: {}'.format(cookie_feq)
mapper = MapReduce(mapper_count, reducer_count)
cookie_feq = mapper(cookie_feq)
cookie_feq.sort(key=operator.itemgetter(1), reverse=True)
for freq, count in cookie_feq:
    print '{0}\t{1}'.format(freq, count)
```

这个 MapReduce 的方式依赖服务器的 CPU 核数。理论上核数越多运行越快。使用这个方案，有 24 核的单个服务器完成一次计算只需要 40 几分钟，利用后面提到的 Cython 还可以再提速 30% 左右。其实很多时候不是 Python 太慢，而是没有选对正确的方法。

使用 DPark

DPark 是一个基于 Mesos 的集群计算框架，是 Spark 的 Python 实现版本，类似于 MapReduce，但是比其更灵活。DPark 有如下特点：

- 它是基于 Python 实现的数据处理平台。
- 快速开发，快速运算，高吞吐量，可以轻松处理海量数据。
- 支持循环迭代计算。DPark 摆脱了传统 MapReduce 平台对迭代计算的限制，利用内存计算资源加速计算。
- 高健壮性，能容忍局部错误。DPark 实现了 RDD (Resilient Distributed Datasets)，在出现局部故障的时候可以自动重试计算，避免整个任务失败。
- 原生对 MooseFS 的支持。DPark 可以智能分析数据分布，来调度计算资源分布，尽可能实现 I/O 本地化。

分布式文件系统 MooseFS

NFS (Network File System) 即网络文件系统，允许网络中的计算机之间通过 TCP/IP 网络共享资源。也就是说，本地 NFS 的客户端应用可以透明地读写位于远端 NFS 服务器上的文件，就像访问本地文件一样。这样做的好处是，既节省本地存储空间，又达到数据共享。但是 NFS 性能很差，当用户访问量大的时候会让 NFS 不堪重负，而且存在单点故障：一旦 NFS 服务器发生故障，所有靠共享提供数据的应用就不再可用。这个时候应使用分布式文件系统：服务器之间的数据访问不再是一对多的关系（1 个 NFS 服务器，多个 NFS 客户端），而是多对多的关系，性能得到提升的同时也解决了单点故障。MooseFS（以下简称 MFS）是分布式文件系统中非常知名的一个方案，选择它是出于以下的原因：

- 使用简单。MFS 的安装、部署和配置都很容易，很快就可以跑起来服务。
- 支持在线扩容。无须停止服务就可以扩容。
- 使用方便。通过挂载映射就能像访问本地文件一样来访问文件服务器资源。

MFS 系统由 4 部分组成。

1. 元数据服务器 (Master): 负责管理所有文件的系统, 保管每一个文件的元数据 (如大小、文件存放位置等)。
2. 数据存储服务器 (Chunkserver): 真正存储用户数据的服务器。存储文件时, 首先把文件分成块, 这些块在数据存储服务器之间复制, 一般有多个数据服务器。
3. 元数据日志服务器 (Metalogger): 负责备份 Master 的变化日志文件, 文件类型为 changelog_ml.*.mfs, 以便于在 Master 出问题的时候接替其进行工作。
4. 客户端 (Client): 使用 MFS 文件系统来存储和访问的主机称为 MFS 的客户端。成功挂接 MFS 文件系统以后, 就可以像以前使用 NFS 一样共享这个虚拟的存储了。

Mesos

Mesos 是一个集群管理器, 用来作为资源统一管理与调度平台, 让你就像使用一台服务器一样使用整个集群。Mesos 有如下特性:

- 可扩展到 10,000 个节点。
- 支持多种应用, 如 Hadoop、Spark、Kafka、Elastic Search 等。
- 具备多资源调度能力, 可调度内存、CPU、磁盘、端口等。
- 提供 Java、Python、C++ 等多种语言的 API。
- 提供一个 Web 界面查看集群状态。

Mesos 由如下 5 个组件组成。

- Mesos-Master: 负责管理各个 Framework 和 Slave, 并将 Slave 上的资源分配给各个 Framework。
- Mesos-Slave: 负责管理本节点上的各个任务 (Task), 比如为各个 Executor 分配资源。
- Framework: 计算框架, 如 Hadoop, Spark 等, 本节将使用 DPark。
- Executor: 执行器, 安装到 Mesos-Slave 上, 用于启动计算框架中的任务。
- Scheduler: 调度器, 通过注册到 Mesos-Master 来获取集群资源。

一般 Framework 调度运行一个任务，遵循如下的流程：

1. SlaveX 向 Master 报告自己的资源状况，比如有 24 个 CPU 和 64 GB 内存可用。
2. Master 使用 Resource Offers（资源供给）实现跨应用细粒度资源共享，如 CPU、内存、磁盘、网络等。Master 根据公平共享、优先级等策略来决定分配多少资源给 Framework，发送给 FrameworkX，其中描述 SlaveX 有多少可用资源。
3. FrameworkX 中的 Scheduler 会答复 Master 有什么任务需要运行在 SlaveX，以及每个任务需要什么样的资源，比如需要 2 个 CPU、1GB 内存。
4. Master 发送这些任务给 SlaveX。如果 SlaveX 还有未使用的 CPU/内存资源，还可以把这些资源提供给 FrameworkY，重复第 2~4 步。

配置 DPark 环境

基于 MooseFS + Mesos 搭建一个 Dpark 环境，服务器架构如表 11.1 所示。

表 11.1 服务器架构

Server	MooseFS 角色	Mesos 角色
192.168.1.230	Master/Client	Mesos-Master
192.168.1.231	Chunkserver/Client	Mesos-Slave

测试环境，没有用到 Metalogger。

在 DPark 项目源码的 docker 目录下存放了一些 Docker 配置，但是为了增加通用性，使用最新的 MooseFS 和 Mesos。由于编译安装 Mesos 会占用大量内存，所以需要开启交换分区充当内存。

首先搭建 DPark 通用环境：

```
> sudo apt-get -y install build-essential python-dev python-boto libcurl4-nss-dev
libssl2-dev libssl2-modules maven libapr1-dev libsvn-dev
> cd /srv
```

安装最新版 MooseFS：

```
> wget http://ppa.moosdfs.com/src/moosefs-3.0.74-1.tar.gz
> tar xzf moosefs-3.0.74-1.tar.gz
> cd moosefs-3.0.74
> ./configure
> sudo make install
> cd -
```

安装最新版 Mesos:

```
> wget http://mirrors.cnnic.cn/apache/mesos/0.28.1/mesos-0.28.1.tar.gz
> tar xzf mesos-0.28.1.tar.gz
> cd mesos-0.28.1
> ./bootstrap
> mkdir build
> cd build
> ../configure --disable-java
> sudo make install
> cd -
```

克隆 DPark 代码:

```
> git clone https://github.com/douban/dpark
```

设置 mfs 用户及其他:

```
> sudo pip install -r dpark/docker/base/scripts/requirements.txt
> sudo mkdir /mfs
> sudo useradd -r moosefs
> sudo mkdir -p /var/run/mfs
> sudo chown moosefs.moosefs /var/run/mfs
> sudo ldconfig
> sudo mkdir -p /var/{log,lib}/mesos
```

Mesos 集群管理可以使用 Mesos 提供的方法, 集群设置的文件都存在 `/usr/local/etc/mesos` 目录下, 我们可以改 230 服务器上的设置, 然后这个目录同步到其他 Mesos 服务器上。

```
> cat /usr/local/etc/mesos/masters
192.168.1.230 # 指定Mesos-Master的IP
> cat /usr/local/etc/mesos/slaves
192.168.1.231 # 指定Mesos-Slave的IP, 当前环境下只有一个Slave
> cat /usr/local/etc/mesos/mesos-master-env.sh # 启动Mesos-Master用到的设置
IP=`ifconfig eth1 | head -2 | tail -1 | awk '{print $2}' | cut -c 6-` # eth0是虚拟机
    默认的NAT网卡, 这里新增网卡eth1用来和其他服务器通信
export MESOS_ip=${IP}
export MESOS_log_dir=/var/log/mesos
export MESOS_work_dir=/var/lib/mesos
> cat /usr/local/etc/mesos/mesos-slave-env.sh # 启动Mesos-Slave用到的设置
IP=`ifconfig eth1 | head -2 | tail -1 | awk '{print $2}' | cut -c 6-`
export MESOS_ip=${IP}
export MESOS_master=192.168.1.230:5050
export MESOS_log_dir=/var/log/mesos
export MESOS_work_dir=/var/lib/mesos
```

修改/etc/hosts 文件，添加绑定主机名部分。每个服务器上都需要添加：

```
192.168.1.230 dpark1
192.168.1.231 dpark2
```

现在是 192.168.1.230 的设置：

```
> sudo cp dpark/docker/master/etc/* /etc
> sudo cp dpark/docker/master/mfs/* /usr/local/etc/mfs/
> sudo cp /usr/local/var/mfs/metadata.mfs.empty /var/run/mfs/metadata.mfs
> mknod /dev/fuse c 10 229
> mfsmaster -d
> mfsmount /mfs -H 192.168.1.230
```

作为 Mesos-Master，先设置与其他 Mesos-Slave 的 SSH 信任。需要注意，Ubuntu 默认不允许 root 用户使用 SSH 登录，所以需要修改/etc/ssh/sshd_config 文件，设置“PermitRootLogin yes”，然后重启 sshd。

非常重要的一步，修改主机名，每个服务器都要保证主机名是正确的：

```
> sudo hostname dpark2 # 退出，重新登录后生效，但是需要注意，这样修改的话，重启后会失效
> sudo sh -c "echo 'dpark2' > /etc/hostname" # 保证重启后永久生效
```



如果重启没有生效，还需要添加或者更新/etc/hosts：

```
127.0.1.1 dpark2
```

再使用“sudo reboot”重启就可以了。

然后是 192.168.1.231 的设置：

```
> sudo cp dpark/docker/slave/mfs/* /usr/local/etc/mfs
> sudo mkdir /mfsdata
> sudo chown moosefs.moosefs /mfsdata
> MASTER_IP=192.168.1.230
> mknod /dev/fuse c 10 229
> mfschunkserver -d # 启动mfschunkserver，可以使用Supervisor管理
> sed -i "s/#MASTER#/${MASTER_IP}/g" /usr/local/etc/mfs/mfschunkserver.cfg
> sudo sh -c "echo 'dpark2' > /etc/hostname"
> mfsmount /mfs -H ${MASTER_IP}

> sudo sh -c "echo 'dpark2' > /etc/hostname"
> sudo hostname dpark2
```

同样需要退出，再重新登录，至此配置完成。

在 Mesos-Master 上就可以使用 Mesos 自带的脚本管理集群了。启动集群：

```
> sudo mesos-start-cluster.sh
```

如果想关闭的话就使用：

```
> sudo mesos-stop-cluster.sh
```

需要注意，每个想要运行的 DPark 的服务器上都要有/etc/dpark.conf 配置。配置如下：

```
> cat /etc/dpark.conf
MESOS_MASTER = "192.168.1.230:5050"
MOOSEFS_MOUNT_POINTS = {
    '/mfs': '192.168.1.230'
}
MOOSEFS_DIR_CACHE = True
MEM_PER_TASK = 200.0 # 默认一个任务使用1GB内存，但是通过配置文件可以修改这个值，
                      现在设置为200 MB
```

从 WordCount 开始

假设现在有一个几 GB 的文本文件，想统计文件中出现的频率最高的前三个单词。使用 Python 可以这样实现：

```
result = defaultdict(int)
with open('/mfs/sample.txt') as f:
    for line in f:
        for word in filter(None, line.rstrip().split(' ')):
            result[word] += 1
print sorted(result.items(), key=lambda (x,y):y, reverse=True)[:3]
```

如果使用 DPark，则可以用如下代码实现（wordcount.py）：

```
from dpark import DparkContext

dpark = DparkContext()

def parse(line):
    for word in filter(None, line.rstrip().split(' ')):
        yield word, 1

print dpark.textFile('/mfs/sample.txt') \
    .flatMap(parse) \
    .reduceByKey(lambda x,y: x+y) \
    .top(3, lambda (x,y):y)
```

假设这个文件的内容是：

```
Hello World
Hello Python
```

我们来解析下这 4 行代码。

1. `dpark.textFile` 是从 MFS 上找到文件，读取文件或者目录。它可以并行读取已经分块压缩的文件。

2. `flatMap` 把内容根据设置的解析函数（这里是 `parse` 函数）拆成了单词：

```
In : f = open('/mfs/sample.txt')
# makeRDD/parallelize将本地内存中的list变成一个RDD
In : b = dpark.parallelize(f.readlines())
In : f = b.flatMap(parse)
In : f.collect() # 使用LocalScheduler来本地执行
Out: [('Hello', 1), ('World', 1), ('Hello', 1), ('Python', 1)]
```

3. `reduceByKey`：根据键对结果合并。

```
# 相当于从一个RDD变为另外一个RDD的函数
In : r = f.reduceByKey(lambda x,y: x+y)
In : r.collect()
Out: [('Python', 1), ('World', 1), ('Hello', 2)]
```

4. `top`：按值的数量排序，取数量最多的前 N 个。

```
In : r.top(2, lambda (x, y): y) # 取前2个
Out: [('Hello', 2), ('Python', 1)]
```

使用 Mesos 集群运行它：

```
> python wordcount.py -m mesos
```

其中 “-m/--master” 用来指定计算集群的位置。它支持 5 种模式。

- `local`：使用本地模式，不加参数的默认模式。
- `process`：使用本地多进程模式。
- `mesos`：根据配置文件或 `MESOS_MASTER` 环境变量寻找计算集群。
`export MESOS_MASTER=zk://zk1:2181,zk2:2181,zk3:2181/mesos_master`
- `host[:port]`：根据 ip 和 port 来指定集群位置。
- `zk://hosts/path`：根据 Zookeeper 集群地址和路径来寻找计算集群。

其他常用参数如表 11.2 所示。

表 11.2 其他常用的参数

参 数	含 义
-M/-mem	指定每个任务默认使用的内存大小。如不指定，则默认是 200 MB
-p/-parallel	指定每个计算节点上最多可以同时执行多少任务，可以用来限制任务并发程度
-g/-group	只在指定的计算节点组上运行



你可能遇到浮点数问题，比如 CPU 要求 1.01 个，超出了真实的 1 个，造成任务卡住，那么可以添加 `-c` 参数，设定需要使用的 CPU 数，如：

```
> python wordcount.py -m dpark1:5050 -c 0.5
```

还可以通过 Dpark 使用指南 (<http://bit.ly/23gduHc>) 获取其他常用函数的用法，其中还有开发的注意事项、性能调优方面的指导。DPark 项目的 `examples` 下有一些常见需求的例子，可以用来作为进一步资料了解。`dgrep` 和 `tools` 目录下的 `drun` 是非常好用的工具。`drun` 把任务分配到各计算节点来运行，避免大家的程序都拥堵在同一台机器运行。如果任务和服务器节点无关，可以开启。至于 `dgrep`，如果 `grep` 的目标是共享内容（如在 NFS 上），可以让各计算节点一起来运行。

PV & UV 统计

PV/UV 是网站分析中最基础、最常见的指标。PV 是 Page View 的缩写，表示页面浏览量，UV 是 Unique Visitor 的缩写，表示独立访客量。

网站都会记录多种类型的日志，如用来做数据分析、算法推荐、产品决策等。假设现在有一种类型的日志，记录了用户 ID、访问的 URL、浏览器类型、访问时间等。日志的格式如下：

```
2016/06/08\t11:39:59\t31632288\twww.dongwm.com/movie/12345\t22\t200\t1349387474\twww.google.com\t2
```

字段之间用 Tab 分开，其含义如表 11.3 所示。

表 11.3 日志中的字段及其含义

字 段	含 义
date	访问日期
time	访问时间
uid	用户 ID

续表

字 段	含 义
url	访问的 URL
browser_type	浏览器类型，它由用户代理转换获得
statue_code	访问返回的状态码
encrypted_ip	转换为整数的 IP
url_referer	本次访问的 Referer 来源
access_type	访问类型，类型包含移动设备（1）、PC（2）、爬虫（3）等

通常每次都会有多个 PV/UV 需求, 日期一般是一个范围, 假如每个需求都使用 DPark 运行一遍的话未免效率太低下了（因为这是在重复地解析相同的日志）。可以通过如下方式一次性获得多种 PV/UV 的数据。

首先定义一个去除异常日志的函数, 去掉状态码大于或等于 400、访问类型大于或等于 3 的访问记录:

```
def exclude_unusual(f):
    @wraps(f)
    def wrapper(log, *a, **kw):
        if int(log.access_type) < 3 and int(log.statue_code) < 400:
            return f(log, *a, **kw)
        return []
    return wrapper
```

为了方便地使用日志属性, 使用 collections.namedtuple 定义了一个 Weblog 的类:

```
import collections

_Weblog = collections.namedtuple('Weblog', [
    'date', 'time', 'uid', 'url', 'browser_type', 'statue_code',
    'encrypted_ip', 'url_referer', 'access_type'
])

class Weblog(_Weblog):
    @classmethod
    def from_line(cls, line):
        fields = line.strip().split('\t')
        return cls(*fields)
```

每条日志都可以转化为 Weblog 实例, 如上面的那条记录, log.date 的值就是 “2016/06/08”, log.browser_type 的值就是 22。

先给 UV 和 PV 定义 Runner 基类，添加一些共用的方法：

```
class BaseRunner(object):
    def __init__(self, log_path=None, date_=None, match_rules=None):
        if log_path is None:
            log_path = '/mfs/log/web_log'
        self.date = [] if date_ is None else date_
        self.match_rules = {} if match_rules is None else match_rules
```

通过 self.date 传入日期范围，而通过 self.match_rules 传入多个需要执行的规则。

_filter_func 方法收集了全部的过滤规则，但是作为装饰器传入，在 Dpark 中才执行：

```
def _filter_func(self):
    filter_func_list = self.match_rules.values()

    def wrapper(log, *args, **kwargs):
        return any(func(log, *args, **kwargs)
                    for func in filter_func_list)

    return wrapper
```

通过下面两个方法获得日期范围内的全部日志文件列表：

```
import glob
import itertools

def _get_paths_by_date(self, date_):
    return glob.glob(os.path.join(
        self.log_path, '*', '*', '*', date_.strftime('%Y%m%d')
    ))

@property
def paths(self):
    return itertools.chain(self._get_paths_by_date(date_)
                           for date_ in self.date)
```

然后把 Dpark 的用法封装起来：

```
def get_rdd(self):
    dpark = DparkContext()

    return dpark.union(
        [dpark.textFile(path, splitSize=64 << 20)
         for path in self.paths]
    ).map(Weblog.from_line)

def get_flat_mapped_rdd(self):
```



```

filter_func = self._filter_func()
map_func = self._map_func()
return (self.get_rdd()
        .filter(filter_func)
        .map(map_func)
        .flatMap(lambda x: x))

```

其中 `_map_func` 由于 PV/UV 的归纳方式不同，需要在子类中实现；`get_flat_mapped_rdd` 方法抽象了 PV/UV 使用 DPark 的相同部分。

PVRunner/UVRunner 的实现如下：

```
class PVRunner(BaseRunner):
```

```

    def _map_func(self):
        def wrapper(log, *args, **kw):
            values = []
            for k, v in self.match_rules.iteritems():
                if v(log, *args, **kw):
                    values.append((k, 1))
            return values
        return wrapper

    def get_result(self):
        flat_mapped_rdd = self.get_flat_mapped_rdd()
        return (flat_mapped_rdd.reduceByKey(lambda x, y: x + y)
                .collectAsMap())

```

```
class UVRunner(BaseRunner):
```

```

    def _map_func(self):
        def wrapper(log, *args, **kw):
            values = []
            for k, v in self.match_rules.iteritems():
                if v(log, *args, **kw):
                    values.append((k, log.uid, 1))
            return values
        return wrapper

    def get_result(self):
        flat_mapped_rdd = self.get_flat_mapped_rdd()
        return (flat_mapped_rdd.uniq()
                .map(lambda x: (x[0], 1))
                .reduceByKey(lambda x, y: x + y)
                .collectAsMap())

```

现在需要统计访问/subject/、/item/、去掉异常访问的/subject/这三种数据，在 2016 年 5 月 22 日的 PV 数，以及 2016 年 5 月 22 日到 2016 年 5 月 27 日（包含）的 UV 数。可以这样写：

```
from tyrion import PVRunner, UVRunner, exclude_unusual

def is_subject(log):
    return 'show/' in log.url

def is_item(log):
    return 'item/' in log.url

match_rules = {
    'item': is_item,
    'subject': is_subject,
    'excluded_subject': exclude_unusual(is_subject)
}
runner = PVRunner(date(2016, 5, 22))
runner.match_rules = match_rules
pv = runner.get_result()

date_range = [date(2016, 5, 22) + timedelta(days=i) for i in range(6)]
runner = UVRunner(date_range)
runner.match_rules = match_rules
uv = runner.get_result()
```

数据报表

Web 开发者的日常工作之一是给运营、产品、客服以及开发团队创建一些数据报表，本节将展示 3 个有用的例子。

发送带有样式和附件的邮件

常见需求之一是给运营和产品同事定期发送一些用于运营和产品决策的统计邮件。邮件通常是以表格为主体，一般也会带有一个或者多个附件。数据报表中的数据种类太多的话，如果不格式化处理，读起来很痛苦，也不容易从中获取有用信息。其实我们可以让邮件带有 CSS 样式，让读邮件也成为一件愉快的事情。

现在有这样一需求：运营人员维护了一些核心用户池（如发布东西图文、海淘经验等），希望每周一看到上一周这些核心用户的一些数据。下面的例子展示了 10 个发布图文的核心用户上周每天的发布量（email_with_mako.py），它做了如下 4 件事：

- 把数据写进 csv 文件。
- 使用 Mako 把数据渲染成邮件内容。
- 生成邮件。
- 发送带附件的邮件。

我们先定义用到的假数据和 main 函数：

```
rows_data = [
    [34, 72, 38, 30, 75, 48, 75],
    [6, 24, 1, 84, 54, 62, 60],
    [28, 79, 97, 13, 85, 93, 93],
    [27, 71, 40, 17, 18, 79, 90],
    [88, 25, 33, 23, 67, 1, 59],
    [24, 100, 20, 88, 29, 33, 38],
    [6, 57, 88, 28, 10, 26, 37],
    [52, 78, 1, 96, 26, 45, 47],
    [60, 54, 81, 66, 81, 90, 80],
    [70, 5, 46, 14, 71, 19, 66],
]
col_headers = ['日期', '周一', '周二', '周三',
               '周四', '周五', '周六', '周日']
row_headers = ['用户{}'.format(i) for i in range(1, 11)]

def main():
    csv_file = os.path.join(HERE, 'statistics.csv')
    tmpl_directories = [os.path.join(HERE, 'tmpl')]
    write_csv(csv_file, col_headers, rows_data)
    data = {'rows_data': rows_data, 'row': col_headers,
           'row_headers': row_headers}
    content = mako_render(data, 'statistics.txt', directories=tmpl_directories)
    sendmail(content, '核心用户运营数据', [csv_file], nick_from='豆瓣网')
```

现在开始分解 main 函数中未实现的功能。先看生成 csv 附件的函数：

```
import csv

def write_csv(csv_file, headers, rows):
    f = open(csv_file, 'wt')
    writer = csv.writer(f)
    writer.writerow(headers)
    for index, row in enumerate(rows):
        writer.writerow([row_headers[index]] + row)
    f.close()
```

再看使用 Mako 模板渲染的函数：

```
from mako.template import Template
from mako.lookup import TemplateLookup

def mako_render(data, mako_file, directories=['tpl']):
    mylookup = TemplateLookup(directories=directories, input_encoding='utf-8',
                              output_encoding='utf-8',
                              default_filters=['decode.utf_8'])
    mytemplate = Template('<%include file="{}/>'.format(mako_file),
                           lookup=mylookup, input_encoding='utf-8',
                           default_filters=['decode.utf_8'],
                           output_encoding='utf-8')
    content = mytemplate.render(**data)
    return content
```

Mako 默认会把传入的字符串作为函数 `unicode` 的参数执行，由于文本中有中文，会出现 `UnicodeDecodeError` 错误，所以一定要使用 “`default_filters=['decode.utf_8']`”。

接下来引入构造和发送邮件的模块：

```
import smtplib
from email.header import Header as _Header
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.utils import parseaddr, formataddr
```

其中 `email` 负责构造邮件，`smtplib` 负责发送邮件。还需要定义发送邮件相关的常量：

```
SMTP_SERVER = 'smtp.qq.com'
SMTP_PORT = 587
FROM_ADDR = 'xxx@qq.com'
PASSWORD = 'password'
TO_ADDRS = ['yy@gmail.com']
```

使用标准的 25 端口连接 SMTP 服务器是明文传输，发送过程中可能会被窃听。推荐选择 STARTTLS 协议的 587 端口，它能够让明文的通信连接直接成为加密连接（使用 SSL 或 TLS 加密）。

构造一个邮件通常需要定义发送者（From）、接收者（To）、主题（Subject），以及附件（可以有多个），所以构造邮件的函数比较复杂：

```
def gen_msg(content, subject, attachments, nick_from=None, nick_to='运营'):
    if nick_from is None:
        nick_from = FROM_ADDR
    msg = MIMEMultipart()
```

```

msg['From'] = _format_addr('{ } <{}>'.format(nick_from, FROM_ADDR))
msg['To'] = _format_addr('{ } <{}>'.format(nick_to, TO_ADDRS))
msg['Subject'] = Header(subject)
msg.attach(MIMEText(content, 'html', 'utf-8'))

for attachment in attachments:
    attach = MIMEText(open(attachment, 'rb').read(), 'base64', 'utf-8')
    attach['Content-Type'] = 'application/octet-stream'
    attach['Content-Disposition'] = 'attachment; filename="{ }"'.format(
        os.path.basename(attachment))
    msg.attach(attach)
return msg

def Header(name): # noqa
    return _Header(name, 'utf-8').encode()

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name), addr))

```

上面使用了 `_format_addr` 格式化邮件地址来支持中文 `<addr@example.com>` 这样的带有昵称的邮件地址。用函数 `Header` 伪装成 `Header` 类后就可以使用 `Header(name)` 代替较长的 `Header(name, 'utf-8').encode()` 了。

最后一步就是发送邮件了：

```

def sendmail(content, subject, attachments, nick_from=None):
    msg = gen_msg(content, subject, attachments, nick_from)
    server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
    server.starttls()
    server.login(FROM_ADDR, PASSWORD)
    server.sendmail(FROM_ADDR, TO_ADDRS, msg.as_string())
    server.quit()

```

现在各邮件服务商对邮件的使用样式管得比较严，最开始在邮件的 HTML 内容中加 link 标签就可以，现在这种方式普遍不可用了。另外的方法是把样式文件当作附件，这样就可以在邮件内容中引用附件来显示样式，缺点是附件中多出来的样式文件，会让需求方很迷惑。本例是直接把样式写进了 HTML 的 style 标签中，为了节省篇幅，使用 Bootstrap 框架的样式来演示，现在已经下载了 `bootstrap.min.css`，但实际工作中不应该把整个 Bootstrap 样式文件都插入到内容中，这会增加差不多 100 KB 的内容。应该只保留少量用到的样式。

邮件模板使用 include 添加 CSS 样式：

```
<style>
  <%include file="bootstrap.min.css"/>
</style>
<table class="table table-bordered table-hover">
  <tr>
    % for column in row:
      <th>${column}</th>
    % endfor
  % for index, rows in enumerate(rows_data):
    <tr>
      % for column in [row_headers[index]] + rows:
        <td>${column}</td>
      % endfor
    </tr>
  % endfor
</table>
```

发送一封邮件后，在 Mac 自带的邮件客户端 Mail 里就可以看到收到的邮件的效果了，如图 11.1 所示。

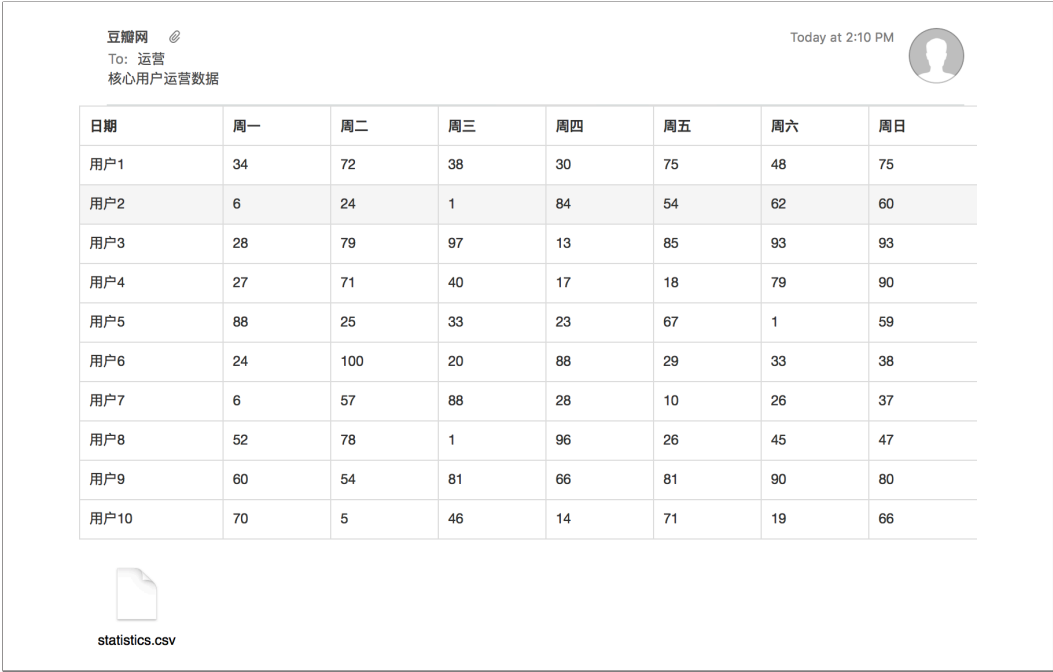


图 11.1 收到的邮件

创建 xlsx 文件

.xlsx 是 Microsoft Office Excel 从 2007 版本开始使用的文档扩展名。除非报表中只包含简单的字符串和数字，否则不建议使用 csv 格式，因为虽然 csv 格式对开发者友好也容易实现，但是 csv 对于使用 Windows 系统的非技术人员来说并不友好，常见的就是乱码问题。本节将向大家展示创建两个复杂的 xlsx 文件例子。

创建 xlsx 时使用了 XlsxWriter，我们先安装它：

```
> pip install XlsxWriter
```

包含带样式和 Sparkline 图表

从 Excel 2010 开始，Excel 增加 Sparkline 功能。它是一种信息体积小、数据密度高的图表，是研究数据走势的一个重要依据。本例将使用上例的需求和假数据来完成（core_user_report.py）：

```
from __future__ import unicode_literals
import xlsxwriter

from email_with_mako import rows_data, col_headers, row_headers

# 创建一个叫作core_user_report.xlsx的文件
workbook = xlsxwriter.Workbook('core_user_report.xlsx')
# 添加一个工作表，名字为“核心用户”
worksheet = workbook.add_worksheet('核心用户')
# add_format用来创建样式，支持的样式列表详见
bold = workbook.add_format({'bold': True, 'align': 'center'}) http://xlsxwriter.
    readthedocs.org/format.html

format1 = workbook.add_format({'bg_color': '#FFC7CE',
                                'font_color': '#9C0006'})
format2 = workbook.add_format({'bg_color': '#C6EFCE',
                                'font_color': '#006100'})

for index, header in enumerate(row_headers, 2):
    col = 'A{}'.format(index)
    worksheet.write(col, header, bold) # 设置竖轴

for row, row_data in enumerate(data):
    worksheet.write_row(row + 1, 1, row_data)

# 条件格式化，在左上角B2到右下角H11的区域内，如果值大于等于50，使用样式format1，
    否则使用样式format2
worksheet.conditional_format('B2:H11', {'type': 'cell',
```

```

        'criteria': '>=',
        'value': 50,
        'format': format1})

worksheet.conditional_format('B2:H11', {'type': 'cell',
        'criteria': '<',
        'value': 50,
        'format': format2})

# I列是Sparkline图表所在的列，加大它的宽度
worksheet.set_column('I:I', 20)
for i in range(2, 12):
    # 根据每一排的7天数据生成Sparkline图表
    worksheet.add_sparkline(
        'I{}'.format(i),
        {'range': '核心用户!B{0}:H{0}'.format(i), 'markers': True,
         'series_color': '#E965E0'})

workbook.close() # 需要显式地关闭workbook

```

执行脚本后在当前目录下就生成了 `core_user_report.xlsx` 文件。但由于 Mac 自带的 Numbers 不支持 Sparkline 功能，会忽略 Sparkline 图表，图 11.2 所示的是从 Windows 上打开文件的效果。



图 11.2 从 Windows 上打开文件的效果



因为 `xlsxwriter` 要求值的类型是 `unicode`，在一开始使用了 “`from __future__ import unicode_literals`”，这样相当于自动把字符串转换为 `unicode`。

包含多工作表和图表

下面的例子将统计上周每天一些活动页面的 PV/UV，两种指标放在不同工作表中，并分别生成折线图和柱状图（pv_uv.py）。

```
from __future__ import unicode_literals

import xlswriter

workbook = xlswriter.Workbook('pv_uv.xlsx')

worksheet1 = workbook.add_worksheet('pv')
worksheet2 = workbook.add_worksheet('uv')
# 这是创建的第三个工作表，只为演示，没有数据
worksheet3 = workbook.add_worksheet('analysis')

chart1 = workbook.add_chart({'type': 'line'}) # PV使用折线图
chart2 = workbook.add_chart({'type': 'column'}) # UV使用柱状图

worksheet1.set_tab_color('#4271ae') # 设置工作表的Tab颜色
worksheet2.set_tab_color('#c82829')
worksheet3.set_tab_color('green')

bold = workbook.add_format({'bold': 1})

pv_data = [[20233, 27855, 30126, 22737, 23331, 34791, 18075],
            [31001, 34483, 33221, 29448, 27082, 31534, 18035],
            [30771, 34543, 30001, 26257, 30778, 22168, 27469],
            [34605, 31545, 26359, 32073, 29603, 32025, 32674],
            [24035, 32267, 19562, 24721, 19573, 31712, 28171]]

uv_data = [[15509, 13787, 14492, 14093, 14008, 10630, 10363],
            [11727, 15526, 15865, 12235, 14798, 10056, 11561],
            [12699, 13125, 15009, 9606, 9555, 17222, 17231],
            [16110, 13798, 10435, 11363, 11862, 10981, 10113],
            [9306, 17165, 9803, 14932, 13226, 13047, 17671]]

cols = ['专题页{}'.format(i) for i in range(1, 6)]
headings = ['专题页', '周一', '周二', '周三', '周四', '周五', '周六', '周日']

for sheet, data in ((worksheet1, pv_data),
                    (worksheet2, uv_data)):
    sheet.write_row('A1', headings, bold)
    sheet.write_column('A2', cols, bold)
    for index, row in enumerate(data, 2):
```

```
sheet.write_row('B{}'.format(index), row)

for sheet, chart in ((worksheet1, chart1),
                     (worksheet2, chart2)):
    for index in range(1, 6):
        chart.add_series({ # 给图表添加数据
            'name': '活动页{}'.format(index),
            'categories': '={0}!$B$1:$H$1'.format(sheet.name),
            'values': '={0}!$B${1}:$H${1}'.format(
                sheet.name, index + 1),
        })

chart1.set_title({'name': '上周活动页 PV'}) # 设置标题
chart1.set_x_axis({'name': 'PV'}) # 设置x轴
chart1.set_y_axis({'name': '数量'}) # 设置y轴
chart1.set_style(10) # Excel 2007开始包含了默认的48种Chart样式, 效果可参见
                    chart_styles.xlsx

worksheet1.insert_chart('A8', chart1, {'x_offset': 25, 'y_offset': 10})

chart2.set_title({'name': '上周活动页 UV'})
chart2.set_x_axis({'name': 'UV'})
chart2.set_y_axis({'name': '数量'})
chart2.set_style(42)
worksheet2.insert_chart('A8', chart2, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

上面的 categories 都使用了 “={0}!\$B\$1:\$H\$1”.format(sheet.name)”, 相当于寻找工作表中对应坐标或者坐标集合的值, 也就是 B1-H1 这 7 个坐标的结果。

效果图如图 11.3 和图 11.4 所示。

使用 Pandas

Pandas 是一个结构化数据分析工具, 它使得在 Python 中处理数据非常快速和简单。Pandas 建造在 NumPy 之上, 对于 Web 开发来说使用 NumPy 的机会不多, 但是在非常多的场景下都可以通过 Pandas 简化我们的工作。本节我们将展示基于 11.4 节产生的数据来做数据分析以及演示如何与 Flask 应用集成。

我们先安装它:

```
> pip install pandas
```

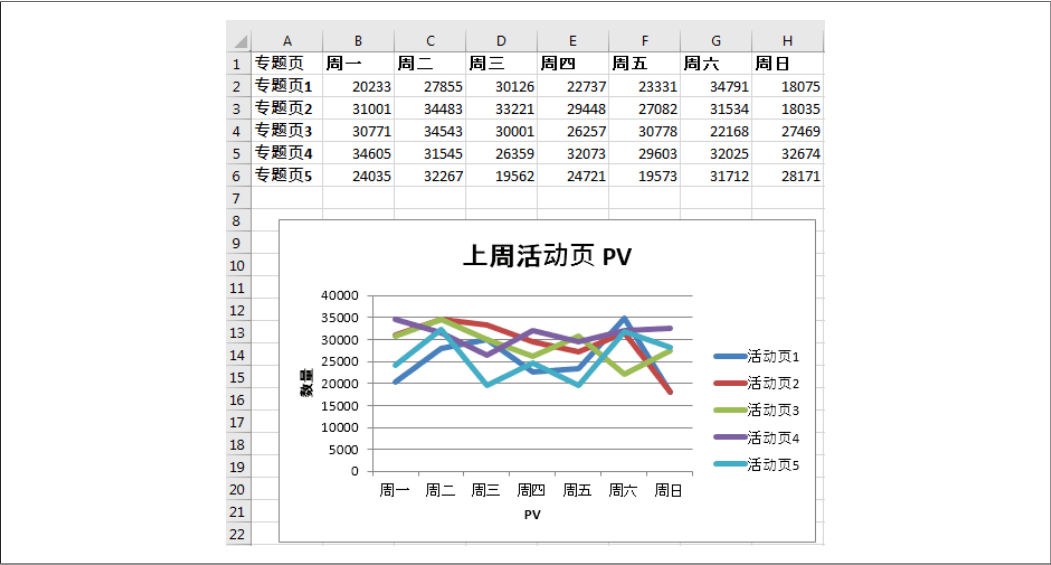


图 11.3 效果图 1

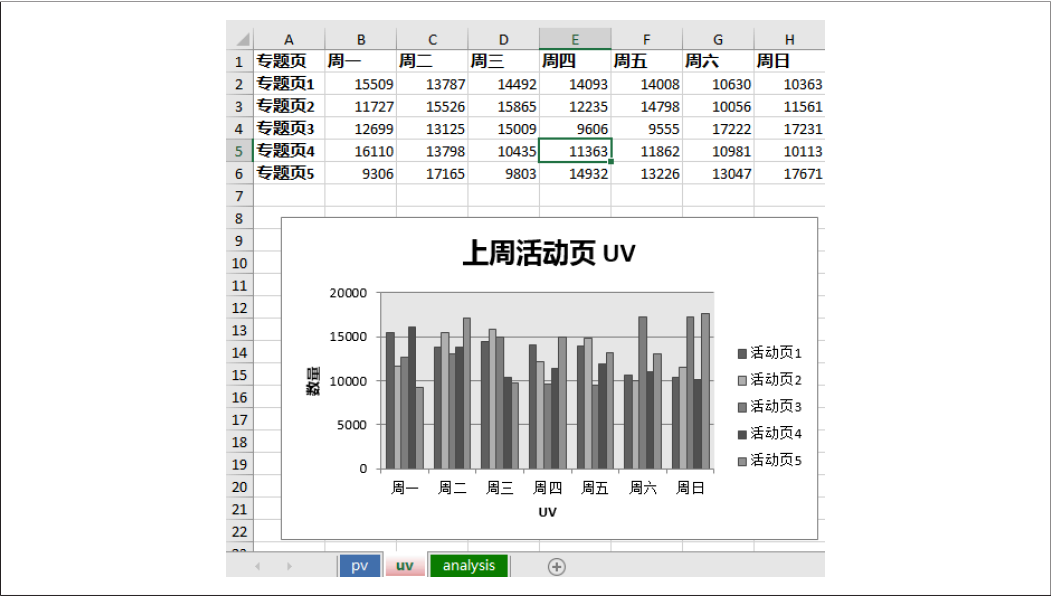


图 11.4 效果图 2

Pandas 入门

Pandas 包含三种数据结构。

1. Series: 一维数组, 与 Python 的列表类似, 但 Series 中则只允许存储相同的数据类型, 这样可以更有效地使用内存, 提高运算效率。Series 的元素还可以自定义索引。

```
In : from pandas import Series
In : s = Series([1, 2, 3])
In : s[1]
Out: 2
In : s = Series([1, 2, 3], index=['a', 'b', 'c'])
In : s.a
Out: 1
```

2. DataFrame: 二维的表格型数据结构, 可以把它想象成 Excel 的工作表, 它既有行索引也有列索引。

```
In : from pandas import DataFrame
In : data = {'a': [1, 2, 3], 'b': ['a', 'c', 'b'], 'c': ['x', 'y', 'z']}
In : df = DataFrame(data)
In : df
Out:
   a  b  c
0  1  a  x
1  2  c  y
2  3  b  z
In : df.a[1]
Out: 2
In : df = DataFrame(data, index=['one', 'two', 'three'], columns=['a', 'b', 'c'])
In : df.a.two
Out: 2
```

3. Panel: 三维的数组, 可以看作是一个三维的 DataFrame, 在 Web 开发中不常用。

本节主要使用 DataFrame 这种数据类型, 它提供如下常用方法。

1. 把 DataFrame 对象转化成字典。

```
In : df.to_dict()
Out:
{'a': {'one': 1, 'three': 3, 'two': 2},
 'b': {'one': 'a', 'three': 'b', 'two': 'c'},
 'c': {'one': 'x', 'three': 'z', 'two': 'y'}}
```

2. 行和列的对调。

```
In : df.T
Out:
   one two three
a    1    2     3
```

```

b   a   c   b
c   x   y   z

```

3. 排序。可以以轴/列来进行排序。`ascending` 是排序方式，默认值为 `True`，即降序排列：

```

# axis是指用于排序的轴，可选的值有0和1，默认值为0，即按行标签（Y轴）排序，若为1则按照列标签排序
In : df.sort_index(axis=0, ascending=True)
Out:
      a  b  c
one   1  a  x
three 3  b  z
two   2  c  y
In : df.sort_values(by='b') # 按照列排序
Out:
      a  b  c
one   1  a  x
three 3  b  z
two   2  c  y
In : df.sort_values(by=['b', 'c'], ascending=[0, 1]) # 可以进行多重排序，by和ascending的值的个数需要相等
Out:
      a  b  c
two   2  c  y
three 3  b  z
one   1  a  x

```

4. 读写数据。`DataFrame` 可以方便地读写数据文件，支持 `csv`、`Excel` 和 `HDF5` 等类型：

```

In : import pandas as pd
In : csv = pd.read_csv('chapter11/section4/statistics.csv')
In : csv.head(1) # head表示列出前N行，默认为5行
Out:
   日期  周一  周二  周三  周四  周五  周六  周日
0  用户1  34  72  38  30  75  48  75
In : csv.to_csv('new.csv') # 把内容保存到new.csv中
# 解析Excel需要预先安装Excel的模块xlrd
In : xlsx = pd.read_excel('chapter11/section4/pv_uv.xlsx', 'uv', index_col=None, na_values=['NA'])
In : xlsx.head(1)
Out:
   专题页  周一  周二  周三  周四  周五  周六  周日
0  专题页1  15509  13787  14492  14093  14008  10630  10363
# 把内容保存到new.xlsx中
In : xlsx.to_excel('new.xlsx', sheet_name='sheet1')

```

读取 MySQL 数据库

Pandas 可以方便地读取 MySQL 中的数据：

```
In : import MySQLdb
In : from pandas.io.sql import read_sql
In : db = MySQLdb.connect('localhost', 'web', 'web', 'r')
In : query = 'SELECT id, filename, filehash from PasteFile'
In : df = read_sql(query, db)
In : item = df.head(1)
In : item
Out:
   id          filename          filehash
0  1  sample.pdf  c4b19d2db0ce4530b7e92a81b52a7a63.pdf
In : item.filehash
Out:
0    c4b19d2db0ce4530b7e92a81b52a7a63.pdf
Name: filehash, dtype: object
In : item.filename.get(0) # 获得数据库对应字段的值
Out: 'sample.pdf'
```

我们还能把数据写进数据库。需要注意的是，con 只接受 SQLAlchemy 引擎对象和 DBAPI2（只支持 sqlite3）：

```
In : from sqlalchemy import create_engine
In : engine = create_engine('mysql://web:web@localhost:3306/r', echo=False)
In : df.to_sql(name='new_table', con=engine, if_exists='append')
```

to_sql 的 if_exists 有 3 个选项。

- fail：如果表存在，则什么都不做。
- replace：如果表存在，先 drop 掉，然后重新创建，再插入数据。
- append：如果表不存在，则先创建表，否则直接插入数据。

这对开发有非常大的帮助，我们可以让数据在 csv、Excel、MySQL 之间通过 DataFrame 结构转换，非常方便。

和 Flask 应用集成

11.3 节数据报表中生成的 Excel 文件找起来很麻烦，还得翻历史邮件。如果使用 Pandas 来开发这样的应用就可以简化需求方的工作，只需要少许的 Flask 开发的经验就可以搭建一个数据报表的应用。

这个应用将提供统一格式的页面风格和访问路径，直接在 Web 页面中就可以看到数据，而且可以下载。为了减少复杂度，CSS 样式依然借助了一部分 Bootstrap 框架。

首先实现将 csv 文件转化为 DataFrame 的通用函数，传入“20160423”这样的字符串就会获得 df 对象：

```
def _get_frame(date_string):
    if date_string is None:
        date_string = date.today().strftime('%Y%M%d')
    else:
        try:
            datetime.strptime(date_string, '%Y%M%d')
        except ValueError:
            return False
    df = pd.read_csv(os.path.join(
        CSV_DIRECTORY, '{}.csv'.format(date_string)))
    return df
```

第一个视图用来渲染某天的 csv 数据：

```
@app.route('/csv/<date_string>')
def show_tables(date_string=None):
    df = _get_frame(date_string)
    if isinstance(df, bool) and not df: # df对象不能直接使用“if not df”
        return 'Bad date format!'
    return render_template(
        'chapter11/csv.html', df=df.to_html(classes='frame'), # 转化为HTML的同时添加
        样式类
        date_string=date_string)
```

这个视图用到的 csv 模板如下 (csv.html)：

```
<!doctype html>
<title>CSV Data</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='stylesheets/
    bootstrap.min.css') }}">
<style>
    table.dataframe, .dataframe th, .dataframe td {
        border: none;
        border-bottom: 1px solid #C8C8C8;
        border-collapse: collapse;
        text-align:center;
        padding: 10px;
        margin-bottom: 40px;
        font-size: 0.9em;
    }
    .frame th {
```

```

        background-color: #00ccff;
        color: white;
    }
    tr:nth-child(odd) {
        background-color:#eee;
    }
    tr:nth-child(even){
        background-color:#fff;
    }
    tr:hover {
        background-color: #aaa;
    }
</style>
<div class="page">
    <h1>时间: {{ date_string }} </h1>
    {{ df|safe }}
</div>

```

这个模板中添加了一些 table 的样式，让 table 的奇数和偶数行的背景色不一样，这样做是为了在看数据的时候更容易集中精力。

第二个视图用于下载 csv 文件，为了演示，下载的只是从 DataFrame 对象获得的单个用户数据：

```

from flask import send_file

@app.route('/csv/download/<date_string>/<int:user_index>')
def serve_csv(date_string=None, user_index=None):
    buffer = cStringIO.StringIO()
    df = _get_frame(date_string)
    if isinstance(df, bool) and not df:
        return 'Bad date format!'
    if user_index is not None:
        df = df.loc[user_index - 1] # 事实上返回的是一个Series
    df.to_csv(buffer, encoding='utf-8')
    buffer.seek(0)
    return send_file(
        buffer, attachment_filename='{}.csv'.format(date_string),
        as_attachment=True, mimetype='text/csv')

```

上面的视图并不会创建临时文件，而是直接在内存中读写。df.loc[index] 表示只取 csv 文件中的某一行。

启动应用后，访问 <http://localhost:9000/csv/20160423> 就可以看到如图 11.5 所示的结果。



图 11.5 直接在 Web 页面就能看到数据

通过 <http://localhost:9000/csv/download/20160423/1> 可以下载 2016 年 4 月 23 日的用户 1 的数据。

第 12 章

帮助工具

开发时要借助第三方的工具帮助我们提高工作效率，趁手的编辑器是一个工具，Git 也是一个工具。本章将介绍和 Python Web 开发有关的工具，主要内容包含：

- 解释为什么应该使用 IPython，配置 IPython、调试复杂代码、并行计算等。
- 介绍 Jupyter Notebook 的用途，配置 Jupyter Notebook，在 Notebook 里使用 Echarts、自定义 JavaScript 和 CSS 样式等高级功能。
- 介绍常用的获取 Linux 服务器相关情况的工具。
- 介绍性能测试工具 Boom 和 tcpcopy，并演示如何搭建一个 tcpcopy 环境。
- 介绍分析 Python 程序性能瓶颈的工具。
- 演示如何定制基于 IPython 的交互解释环境。
- 演示豆瓣东西在 2014 年双十一进行的 Jupyter Notebook 实践。

IPython

Python 工程师需要快速验证代码运行结果是否符合预期。最快捷方便的做法就是使用 Python 自带的交互模式，但是这个 Python Shell 有非常多的弊端：

- 不能在退出时保存历史记录以备未来查询。
- 不支持 Tab 自动补全。

- 不能快速获得模块/函数/类的信息，如参数、文档、原始代码等。
- 不方便在交互环境下执行 Shell 命令。

IPython 是一个基于 Python Shell 的交互式解释器，但是拥有比默认 Shell 强大得多的编辑和交互功能。笔者在开发中，有时候在 IPython 交互环境下的时间甚至比使用编辑器的时间还长。学好 IPython 的必要性不亚于学习编辑器和熟悉 Linux 系统。

2015 年 8 月 12 日发布的 IPython 4.0 做了很多的拆分和组件化工作，IPython 已经分离成了两个组织 IPython 和 Jupyter，Notebook 和其他语言无关的部分（如 QtConsole、Notebook 相关）都以独立的包的方式托管到 Jupyter 组织下；IPython 只保留 IPython Shell 和 Jupyter 内核等相关功能，而且其中核心组件也都拆分成包托管到 IPython 组织下，如 ipykernel、traitlets、ipyparallel 等。

我们先安装 IPython：

```
> pip install ipython
> ipython -V
5.0.0
```

当前 IPython 面向用户的组件主有 5 种，如表 12.1 所示。

表 12.1 IPython 面向用户的组件

组 件	说 明
IPython Shell	交互解释器。也就是直接在终端输入 ipython，就进入了 REPL（Read-Eval-Print-Loop）模式
Jupyter Notebook	网页版的记事本应用，可以在页面上交互地运行程序，通常用作数据可视化，Wiki 系统等
Jupyter Console	使用 Jupyter 协议的 IPython 终端交互解释器
Jupyter QtConsole	一个 Qt 的富编辑器，通常选用跨平台的 PySide 作为 Qt 的 Python 绑定库
IPython ipyparallel	用来进行交互的并行计算



现在安装 IPython 时已经不再安装全部组件了，比如想要使用并行计算功能，就得用如下命令安装：

```
> pip install ipython[parallel]
```

IPython 交互模式

IPython 交互模式下有多个有用的功能。

1. 获得对象信息：输入你想要查看的对象，然后加上一个或者两个问号，就能获得多种对象信息。比如“exit?”，然后回车，就可以看到 exit 函数的文档字符串、文件路径、类型等。如果输入“exit??”回车，就可以看到更多的信息，如对象的源代码。
2. Magic 函数：IPython 有很多 Magic 函数，分为两种类型。一种是 Line magics，单行函数，需要使用“%”开头；另一种是 Cell magics，多行函数或者希望执行其他语言的代码，需要使用“%%”开头。

下面的例子将使用内置的 timeit 函数：

```
In : %timeit range(1000)
100000 loops, best of 3: 9.2 µs per loop
```

```
In : %%timeit x = range(10000)
....: max(x)
....:
```

可以使用“%lsmagic”获得全部可用的 Magic 函数。

3. 调用系统 Shell 命令。只需要在命令前加! 即可：

```
In : !uptime
16:31:11 up 1:41, 3 users, load average: 0.02, 0.06, 0.09
```

4. Tab 自动补全。IPython 可以自动检查对象的属性，通过 object_name.<TAB> 列出全部的子属性，再使用 Tab 切换到对应的属性上，然后回车就可以了。
5. 历史记录。IPython 把输入的历史记录存放在个人配置目录下的 history.sqlite 文件中，并且可以结合 %rerun、%recall、%macro、%save 等 Magic 函数使用。尤为有意义的是，它把最近的三次执行记录绑定在 _、__ 和 ___ 这三个变量上。搜索历史记录时，还支持 Ctrl-r、Ctrl-n 和 Ctrl-p 等快捷键。

常用的 Magic 函数

1. %quickref：可以快速了解 IPython 的功能和用法。
2. %alias：设置别名。

```
In : %alias lg ls -la |grep %s
In : lg Vagrantfile
-rw-r--r-- 1 ubuntu ubuntu 534 Jun 7 10:18 Vagrantfile
```

3. %automagic: Line magics 函数在执行 “%automagic 1” 之后就不需要在使用时输入 “%” 前缀了。
4. %pwd: 作用类似于 Linux 的 pwd 命令, 输出当前目录地址。
5. %cd: 作用类似于 Linux 的 cd 命令, 切换目录。如果不加参数会切换到 Home 目录。
6. %bookmark: 加书签, 如果目录非常常用, 可以为当前目录创建书签, 之后就可以用书签切换目录进来了:

```
In : pwd
/home/ubuntu/web_develop/chapter12
In : bookmark chapter12
In : cd ~
/home/ubuntu
In : cd -b chapter12
(bookmark:chapter12) -> /home/ubuntu/web_develop/chapter12
/home/ubuntu/web_develop/chapter12
```

7. %debug: 激活交互的调试器。

```
In : b = 0

In : 1 / b
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-2-1fcdc364a293> in <module>()
----> 1 1 / b

ZeroDivisionError: integer division or modulo by zero

In : %debug
> <ipython-input-2-1fcdc364a293>(1)<module>()
----> 1 1 / b

ipdb> p b
0
```

8. %edit: 使用编辑器打开, 但需要设定 EDITOR 这个环境变量。假如写了一个很复杂的函数, 代码很长, 执行后发现不符合预期, 用历史记录找到这个函数, 再用鼠标移到对应的位置修改就很不方便。其实这时应该使用 edit 来编辑:

```
In : def a(num):
.....:     return num + 1
.....:
In : edit a
Editing In[12]
```

```
IPython will make a temporary file named: /tmp/ipython_edit_A5CGLB/
ipython_edit_26bIsN.py
Editing... done. Executing edited code...
Out: 'def a(num):\n    return num + 2\n'
In : a(1)
Out: 3
```

9. %hist: %history 的别名，查看历史记录。
10. %load: 把外部代码加载进来。
11. %macro: 把历史记录、文件等封装为宏，以便未来重新执行。

```
In : %hist -n
...
5: x = 1
6: y = 2
7: print x + y
In : %macro my_macro 5-7
=== Macro contents: ===
x = 1
y = 2
print x + y
In : my_macro
3
```

12. %rehashx: 把 \$PATH 中的可执行命令都更新进别名系统，这样就可以在 IPython 中不加感叹号而调用了：

```
In : echo 1
File "<ipython-input-1-334d5669e1fb>", line 1
    echo 1
    ^
SyntaxError: invalid syntax

In : %rehashx
In : echo 1
1
```

13. %timeit: 获得程序执行时间。timeit 是 Python 内置的库，用来测量小代码片的执行时间。

```
In : %timeit pass # 默认执行100,000,000次，显示最快的三次结果
100000000 loops, best of 3: 9.8 ns per loop

In : %timeit -n 100 -r 4 pass
100 loops, best of 4: 9.54 ns per loop
```

14. %save: 把某些历史记录保存到文件中。

```
In : hist -n
...
7: x = 1
8: y = 2
In : save 1.py 7-8
The following commands were written to file '1.py':
x = 1
y = 2
In : !cat 1.py
# coding: utf-8
x = 1
y = 2
```

15. logstart/logoff: 记录会话。退出 IPython 后还可以回到之前的状态。

```
In : logstart # 不指定文件名字则默认使用ipython_log.py
In : a = 1
In : logoff
```

现在可以新开一个 IPython:

```
> ipython -i ipython_log.py
In : a
Out: 1
```

16. %time: 作用类似于 Linux 的 time 命令，计算代码的执行时间。

```
In : %time range(10)
CPU times: user 0 ns, sys: 4 μs, total: 4 μs
Wall time: 8.11 μs
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

配置和自定义 IPython

默认的 IPython 配置目录是 `~/ipython/profile_default`。如果使用了多个 Python 解释器版本或者希望使用不一样的配置，则可以使用如下命令创建配置：

```
> ipython profile create web_develop
```

这样就创建了一个叫作 `web_develop` 的配置。如果要使用特殊的配置，方式如下：

```
> ipython --profile=web_develop
```

`web_develop` 配置目录是 `~/ipython/profile_web_develop`，目录下有一个 `ipython_config.py` 文件，存放 IPython 的配置（默认配置都是注释的）。下面展示一个配置：

```
> cat ~/.ipython/profile_web_develop/ipython_config.py
c = get_config() # 不用担心get_config没有定义
c.TerminalIPythonApp.display_banner = False # 不显示banner
c.InteractiveShell.automagic = True # 让magic函数无须输入前缀“%”就可以执行
c.AliasManager.user_aliases = [ # 添加默认的别名
    ('la', 'ls -al')
]
c.InteractiveShell.colors = 'Linux' # 使用Linux颜色主题
c.InteractiveShell.logstart = True # 自动记录会话
c.TerminalInteractiveShell.confirm_exit = False # 直接退出, 不需要确认
```

IPython 的扩展系统

IPython 有强大的扩展系统, 定义一个扩展非常容易。比较好用的两个内置扩展分别是 autoreload 和 storemagic。storemagic 可以持久化宏、变量和别名。可以添加如下配置到 ipython_config.py 实现自动保存:

```
c.StoreMagics.autorestore = True
```

举个例子, 第一次在 IPython 中执行如下命令:

```
In : l = ['hello', 10, 'world']
In : %store l
Stored 'l' (list)
In : exit()
```

这样就把 l 存储下来了。现在退出 IPython 后重新进入:

```
> ipython
In : l
Out: ['hello', 10, 'world']
```

可以看到, l 能直接使用。我们还可以用这个功能保存一些重要的资源, 这样即使退出 IPython 也能找回来。

autoreload 可以让我们不退出 IPython 就动态修改代码, 在执行代码前 IPython 会帮我们自动重载改动的模块, 这种思想在多种 Web 框架中都可见其踪影。

先看一个简单的例子:

```
> cd ~/web_develop/chapter12/section1
> cat py_autoreload.py
def a():
    return 1
```


在 IPython 里面执行它：

```
In : %load_ext autoreload
In : %autoreload 2
In : from py_autoreload import a
In : a()
Out: 1
```

然后在其他终端上修改 py_autoreload.py，把 a 的返回值改成 2：

```
def a():
    return 2
```

在之前的 IPython 中重新调用 a 函数：

```
In : a()
Out: 2
```

可以看到返回值动态地改变了。

官方扩展索引页（<http://bit.ly/1UxMmRL>）列出了一些扩展，其中也包含笔者写的一个 db.py 的扩展。db.py 是一个使用 Pandas 操作数据的工具。我们先安装它：

```
> pip install ipython-db
```

下载测试用的数据库文件：

```
> wget http://t.cn/R5ow5s6 -O /tmp/baseball.sqlite
```

在 IPython 中加载并使用它：

```
In : %load_ext idb
In : %db_connect sqlite:///tmp/baseball.sqlite
In : %tables allstarfull playerID unique count
Out: 1637
In : df1 = %query select * from allstarfull limit 1;
In : df1
Out:
   playerID  yearID  gameNum      gameID teamID lgID  GP  startingPos
0  aaronha01    1955         0  NLS195507120    ML1   NL    1          None
```

事实上，Pandas 对象在 Jupyter Notebook 文档上的展示效果更好。可以打开 db-example.ipynb（<http://bit.ly/1Uzzivm>）看到效果。

使用 IPython 调试复杂代码

开发人员经常遇到程序执行时报错，如果程序非常复杂，用 pdb 模块调试不方便，一般是通过不断在对应的代码行上加一些 print 语句去调试问题。

假设现在有一个非常简单的脚本（py_debug.py）：

```
a = 1
b = 0
```

```
a / b
```

执行“python chapter12/section1/py_debug.py”肯定会报错，但是无法调试上下文。使用 IPython 会直接执行到出错的地方，并进入 pdb 环境：

```
> ipython chapter12/section1/py_debug.py --pdb
WARNING: InteractiveShell.prompt_out is deprecated, use PromptManager.out_template
-----
ZeroDivisionError                                Traceback (most recent call last)
/home/ubuntu/web_develop/chapter12/section1/py_debug.py in <module>()
      3 b = 0
      4
----> 5 a / b
ZeroDivisionError: integer division or modulo by zero
> /home/ubuntu/web_develop/chapter12/section1/py_debug.py(5)<module>()
      3 b = 0
      4
----> 5 a / b
ipdb> p a
1
ipdb> p b
0
```

双进程模型

Jupyter Console 和 IPython Shell 在终端显示的效果很像，都是一个 While True 的 REPL 模式循环。但是，二者却有本质的区别：直接执行不带参数的 ipython 启动一个进程，而使用 Jupyter Console 启动了两个进程。这个区别也体现在 QtConsole 和 Notebook 上。

这种双进程模型的启动方式都使用了 ZeroMQ 作为消息队列来解耦应用。也就是说，当 Jupyter Console 启动后，还启动一个使用 ipykernel 模块的子进程。主进程用来发送命令和接收从子进程内核返回的结果：

```
ubuntu  8213 3922 0 15:37 pts/2    00:00:05 /home/ubuntu/.virtualenvs/venv/bin/
python /home/ubuntu/.virtualenvs/venv/bin/jupyter-console
ubuntu  8250 8213 0 15:37 ?        00:00:04 /home/ubuntu/.virtualenvs/venv/bin/
python -m ipykernel -f /run/user/1000/jupyter/kernel-8213.json
```

可以开启第三个进程连接这个子进程内核：

```
> jupyter console --existing kernel-8213.json
```

这个新的 Console 和进程 ID 为 8213 的 Console 具备相同的环境。也就是说，你在 ID 为 8213 的 Console 中执行 “a = 1”，那么这个新的 Console 中的 a 也等于 1 了。双进程模型的用途还不止本地不同进程中的互通，还可以把 kernel-8213.json 拷贝到其他服务器上，让其他服务器来连接你本地的 Console 进程。

并行计算

IPython 另一个非常有用的功能是支持并行计算。IPython 的并行计算组件叫作 ipyparallel，从 IPython 4.0 开始被拆分出来了，需要独立安装：

```
> pip install ipyparallel
```

我们先启动计算集群：

```
> ipcluster start -n 4
```

现在启动了 4 个引擎的集群。通过 IPython 连接它：

```
In : import ipyparallel as ipp
In : c = ipp.Client() # 客户端用来向引擎发送任务
In : c.ids # IPython引擎的id列表
Out: [0, 1, 2, 3]
In : %autopx # 开启autopx之后，每个交互模式下的命令会在各自的引擎上执行
In : import os
In : print os.getpid()
[stdout:0] 10282
[stdout:1] 10281
[stdout:2] 10283
[stdout:3] 10280
In : %autopx # 再执行一次就是关闭autopx，因为Magic函数pxconfig需要在autopx关闭时才能用
In : %pxconfig --targets 1 # 指定目标对象，这样下面执行的代码就只会第2个引擎下运行
In : %%px --noblock # 执行异步的代码
....: import time
....: time.sleep(1)
....: os.getpid()
....:
Out: <AsyncResult: execute>

In : %pxresult # 获得异步执行的结果
```

```
Out[1:7]: 10281
In : with c[:].sync_imports(): # 给每个引擎都引入time模块
....:     import time
....:
importing time on engine(s)
In : def f(x):
....:     time.sleep(1)
....:     return x * x
....:
In : v = c[:]
In : v
Out: <DirectView [0, 1, 2, 3]>
In : v.map_sync(f, range(10)) # 同步的执行任务
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
In : view = c.load_balanced_view([1, 2]) # 让并行计算实现负载均衡, 这里指定只使用
    第二个和第三个引擎
In : view
Out: <LoadBalancedView [1, 2]>
In : r = view.map(f, range(10)) # 异步执行任务
In : r
Out: <AsyncMapResult: f>
In : r.ready(), r.elapsed # 查看是否执行结束, 以及花费的时间
Out: (True, 3.023197)
In : r.get() # 获得执行的结果
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

还可以把 view 作为装饰器加在希望并行处理的函数上:

```
In : @view.parallel()
....: def f(x):
....:     return x * x
....:
In : f.map(range(10))
Out: <AsyncMapResult: f>
In : r.get()
Out: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

启动集群后, 在 `~/.ipython/profile_default/security/` 目录下会生成 `ipcontroller-client.json` 文件。将这个文件拷贝到其他服务器上, 就可以在远程调用本机的集群资源了:

```
In : c = ipc.Client('/tmp/ipcontroller-client.json', ssh='192.168.0.210')
```

Jupyter Notebook

Jupyter Notebook（以下简称 JN）就是原来的 IPython Notebook，它是一个基于 Web 的、可以使用 IPython 绝大多数功能的记事本应用。它允许将交互式的代码、图片在 Web 上展示出来，支持不同语言的语法高亮、缩进、Tab 自动补全、在线编辑和保存等功能，广泛用于如下场景：

- 项目说明文档。
- PPT 演示，尤其是交互的数据分析相关的 PPT 演示。
- 个人技术笔记。
- 技术博客。Pelican 通过 pelican-ipy nb 插件以及 Nikola 可以支持 JN 格式文章。

我们先安装它。由于 Notebook 现在已经拆分出来，为了保证能使用全部的功能，需要使用-U 更新全部依赖包：

```
> pip install -U ipywidgets "ipython[notebook]"
```

ipywidgets 是从 IPython 拆分出来的 HTML 挂件项目，需要先激活：

```
> jupyter nbextension enable --py widgetsnbextension
Enabling notebook extension jupyter-js-widgets/extension...
- Validating: OK
```

现在启动它：

```
> cd ~/web_develop/chapter12/section2
> jupyter notebook --port 5000 --no-browser --ip="*"
```

启动 JN 默认会自动打开浏览器访问 Web 应用，使用--no-browser 则会禁止打开。JN 使用的 Web 应用框架为 Tornado。

现在打开“http://localhost:5000”就可以访问 JN 了。首先创建一个 JN 文档。在页面的右侧有一个“New”按钮，单击它，出现下拉框，选择“Python 2”，如图 12.1 所示。

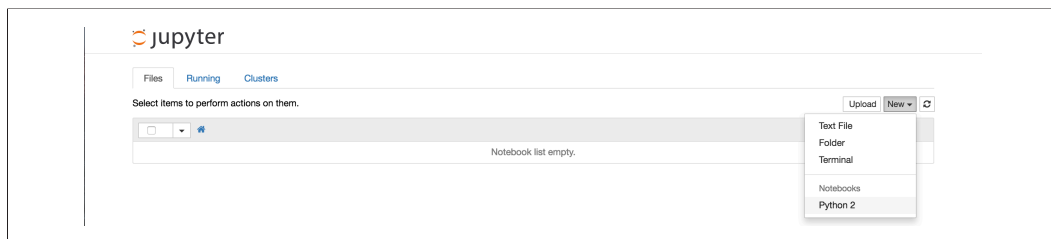


图 12.1 创建一个 JN 文档

新建的文件由于还没有命名，会使用 Untitled.ipynb 这个名字。

默认系统中只有这个“Python 2”内核，也就是支持 Python 2 的语法。如果希望支持更多内核，可以通过官网（<http://bit.ly/23fz8eS>）获得。如果希望同时支持 Python 2 和 Python 3，可以把“Python 3”的内核文件先安装到公用目录/usr/local/share/jupyter/kernels/下：

```
> sudo pip3 install ipython[notebook]
> sudo ipython3 kernelspec install-self
```

图 12.2 所示的是含有 4 个交互执行代码块的例子。

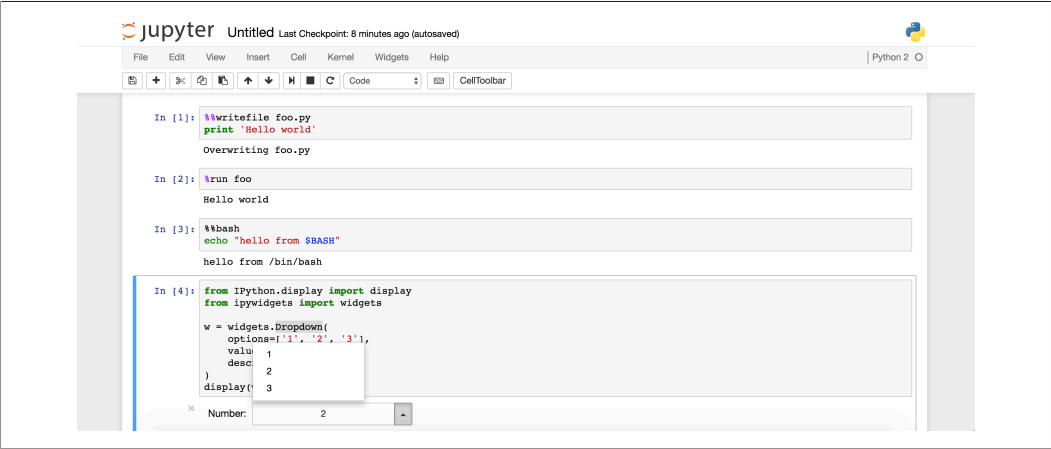


图 12.2 含有 4 个交互执行的代码块

每一个交互区域叫作一个 Cell，前两个使用 IPython 的 Magic 函数执行 Python 代码文件，第三个执行 Bash 代码，第四个是使用自带的下拉菜单（Dropdown）这个 HTML 挂件。页面上可以看到两排菜单：第一排包含了全部的可执行操作，第二排是一些基于第一排菜单中操作的快捷方式。

JN 除了支持代码还支持 Markdown 语法，同时还支持使用键盘快捷键（使用快捷键 H 就可以看到快捷键列表）来执行任务。常用的快捷键如表 12.2 所示。

表 12.2 常用的键盘快捷键

快 捷 键	功 能
B	在当前选择的 Cell 之后添加新的 Cell
A	在当前选择的 Cell 之前添加新的 Cell
Command + S	保存当前的 JN 文件
Shift + Enter	执行当前的 Cell，然后选择下一个 Cell

在单击“New”按钮出现下拉框时，可以看到还有如下两种常见类型。

- 文本文件（Text File）。把 JN 当作在线编辑器，可以在线编辑代码，选择代码语言，编辑时语法高亮，还能保存、重命名、下载、选择编辑器绑定类型（目前支持 VIM、Emacs 和 Sublime Text）等。
- Terminal（终端）。需要安装 terminado 包才能使用这个功能。terminado 是一个为 term.js 提供 Tornado 的 Websocket 后端支持的终端模拟器库，让我们可以把 Web 页面当成终端来使用，如图 12.3 所示。



图 12.3 把 Web 页面当成终端来使用

Notebook 配置

除了在启动时添加参数指定端口和 IP，还可以把对应的设置存放在配置文件中。现在需要手动生成配置文件：

```
> jupyter notebook --generate-config
```

这样就会添加 `~/jupyter/jupyter_notebook_config.py` 文件。我们修改这个文件的如下几个设置：

```
c.NotebookApp.open_browser = False
c.NotebookApp.password = 'sha1:19c53926b1d7:6248a00119f231540f9ba294e2739858da82bba3'
c.NotebookApp.port = 5000
c.NotebookApp.ip = '0.0.0.0'
c.NotebookApp.certfile = '/home/ubuntu/web_develop/chapter12/section2/mycert.pem'
c.NotebookApp.keyfile = '/home/ubuntu/web_develop/chapter12/section2/mykey.key'
```

其中密码并不是明文存储的，而是通过如下命令获得的哈希字符串：

```
In : from notebook.auth import passwd
In : passwd()
Enter password:
Verify password:
Out: 'sha1:19c53926b1d7:6248a00119f231540f9ba294e2739858da82bba3'
```

这个例子中为了加密通信，使用了自建 SSL 证书，通过如下命令获得 pem 和 key 文件：

```
> openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mykey.key -out mycert.pem
```

现在只需要使用“jupyter notebook”启动就可以访问“https://127.0.0.1:5000”了。

Notebook 格式

Notebook 文档有自己的格式，事实上它是一个后缀为.ipynb 的 JSON 文件，包含文本、源码、富媒体输出、Metadata 等类型的数据。我们通过上面的例子生成的 Notebook 文档分析一下 Notebook 格式。

顶级结构包含 cells、nbformat、nbformat_minor、metadata 等 4 个键：

```
{
  "cells": [],
  "metadata": {
    "kernelspec": {
      "display_name": "Python 2",
      "language": "python",
      "name": "python2"
    },
    "language_info": {
      "codemirror_mode": {
        "name": "ipython",
        "version": 2
      },
      "file_extension": ".py",
      "mimetype": "text/x-python",
      "name": "python",
      "nbconvert_exporter": "python",
      "pygments_lexer": "ipython2",
      "version": "2.7.6"
    },
    "widgets": {
      "state": {},
      "version": "1.1.0"
    }
  },
  "nbformat": 4,
  "nbformat_minor": 0
}
```

nbformat 和 nbformat_minor 表示 Notebook 格式版本，一般不需要担心不同版本间的兼容问题，JN 会提示你是否转换格式，在确认后会帮你转换。cells 列表中每个 cell 就是一个交互

执行段。我们展示其中一个 Cell:

```
{
  "cell_type": "code",
  "execution_count": null,
  "metadata": {
    "collapsed": true
  },
  "outputs": [],
  "source": [
    "# %%writefile foo.py\n",
    "print 'Hello world'"
  ]
}
```

cell_type 除了是 code 外还可以是 markdown 等类型, source 就是在 input 框内输入的内容, outputs 是执行的输出。

nbformat 现在也是一个独立的包,通过这样的 Notebook 文档就可以在任何地方生成 HTML 页面:

```
In : import nbformat
In : nb = nbformat.read('/home/ubuntu/web_develop/chapter12/section2/Untitled.ipynb',
    as_version=4)
In : nb['cells'][1]['source']
Out: u'%run foo'
In : nb_v3 = nbformat.convert(nb, 3) # 转换V3版本格式
```

Notebook 格式转换和预览

JN 支持把 Notebook 文档转换为其他格式的文档, Nbconvert 提供这样的转换方法。目前支持的常见格式如下:

- HTML
- LaTeX
- PDF
- Markdown
- ReStructured Text

我们先安装它:

```
> pip install nbconvert
```

假如转换为 HTML, 则执行如下命令:

```
> jupyter nbconvert --to html Untitled.ipynb
```

就会生成同名的 Untitled.html 文件了。

我们还可以定义转换的模板，nbconvert 默认使用了 full.tpl (<http://bit.ly/1UScC88>) 模板，模板引擎是 Jinja2，我们可以使用 Jinja2 语法基于它扩展新的模板：

```
> cat web_develop.tpl
{% extends 'full.tpl' -%}

{% block footer %}
F00000000TEEEER
{% endblock footer %}
```

在转换的时候指定使用 web_develop.tpl 这个模板：

```
> jupyter nbconvert --to html --template web_develop.tpl Untitled.ipynb
```

如果你希望共享 ipynb 文件，可以将其上传到 Nbviewer (<http://nbviewer.jupyter.org>) 网站上，这样就生成了在线的 HTML 页面。它的原理其实很简单，通过如下代码就可以实现：

```
In : from urllib import urlopen

In : import nbformat
In : from nbconvert import HTMLExporter

In : response = urlopen('/path/tp/xxx.ipynb').read().decode()
In : nb = nbformat.reads(response, as_version=4)
In : html_exporter = HTMLExporter()
In : html_exporter.template_file = 'basic'
In : (body, resources) = html_exporter.from_notebook_node(nb)
```

其中 body 就是生成的 HTML 代码。如果不希望你的 JN 文档暴露到外网，可以在内部使用 nbviewer (<https://github.com/jupyter/nbviewer>) 搭建生成静态 HTML 页面的服务。

为什么使用 RequireJS

JN 使用了 RequireJS 作为 JavaScript 文件和模块加载器。在网站发展的早期，JavaScript 代码并不多，完全可以写在一个文件里面，只要加载这一个文件就够了。但是随着业务发展，代码越来越多，就会把它拆分成多个文件，依次加载。通常采用如下的写法：

```
<script src="1.js"></script>
<script src="2.js"></script>
<script src="3.js"></script>
...
```

但是这样做有很大的缺点：

- 加载时浏览器会停止网页渲染，加载的文件越多，网页失去响应的时间就会越长。
- JavaScript 之间可能存在依赖关系，因此必须严格保证加载顺序（比如上例的 1.js 要在 2.js 的前面），当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

使用 RequireJS 就是为了解决这两个问题而诞生的，它能：

- 实现 JavaScript 文件的异步加载，避免网页失去响应。
- 管理模块之间的依赖性，便于代码的编写和维护。

在 Notebook 里使用 Echarts

Echarts 是百度开源的，功能强大、美观、图表丰富、兼容绝大多数浏览器，支持移动设备访问的纯 JavaScript 图表库。它和 Highcharts 以及 amCharts 是三种最常用的图表库。

笔者曾经分享过 Highcharts (<http://bit.ly/1W2V9fM>) 的例子，本节就来使用 Echarts。首先通过 GitHub 的 API 获得 Django 项目的提交活动情况：

```
from collections import defaultdict
from datetime import datetime

import requests
from IPython.display import HTML

r = requests.get('https://api.github.com/repos/django/django/stats/commit_activity')
series = defaultdict(int)
monthname = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
             'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

for d in r.json():
    month = datetime.fromtimestamp(d['week']).month
    series[month] += d['total'] # 数据是按月分开的
```

由于 Echarts 和 Notebook 的加载机制有冲突，需要使用 require.config 配置路径：

```
HTML('''
<div id="chart" style="width: 600px; height:400px;"></div>

<script>
    require.config({
        paths: {
            echarts: '//cdn.bootcss.com/echarts/3.0.0/echarts.min',
        }
    })
</script>''')
```

```
});
require(['echarts'], function(ec) {
    var myChart = ec.init(document.getElementById('chart'));
    var option = {
        title: {
            text: 'Last Year of commit activity',
            left: 'center'
        },
        subtitle: {
            text: 'https://github.com/django/django',
        },
        xAxis: {
            data: %s
        },
        yAxis: {},
        series: [{
            name: 'Commits',
            type: 'line',
            data: %s
        }]
    };

    myChart.setOption(option);
});
</script>
''' % (monthname, series.values()))
```

执行这个 Cell 的结果如图 12.4 所示。这样就方便地实现使用 Notebook 的图表展示了。

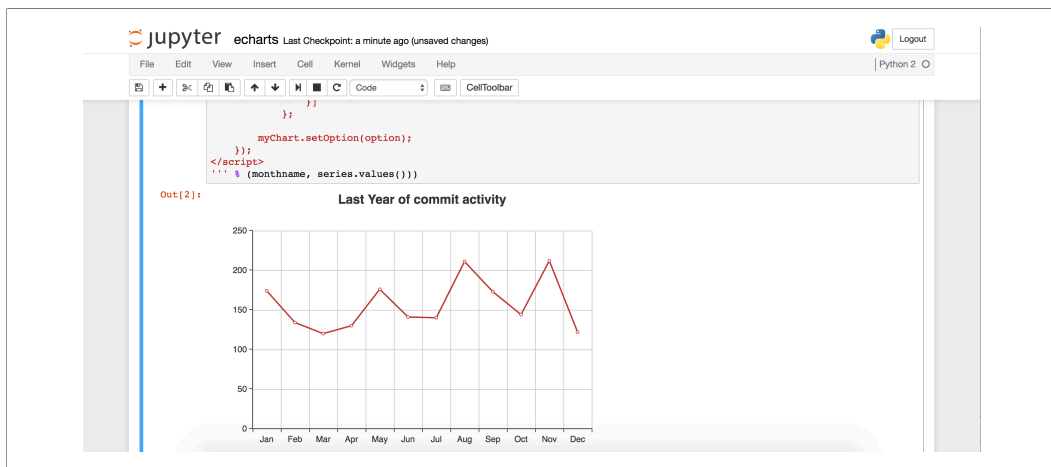


图 12.4 执行 Cell 的结果

富显示

Notebook 除了能展示 HTML 代码，还可以直接显示图片、链接等类型的内容，而且可以定制显示的效果。

直接显示图片时，还可以定义长和宽：

```
from IPython.display import display_html, Image, FileLink, FileLinks, HTML
Image(filename='/tmp/dongxi.jpg', width='100px')
```

使用 FileLink 和 FileLinks 可以显示文件链接：

```
FileLink('rich-display.ipynb')
FileLinks('/tmp')
```

Notebook 可以自动把表格用可视化的样式显示出来，如图 12.5 所示。

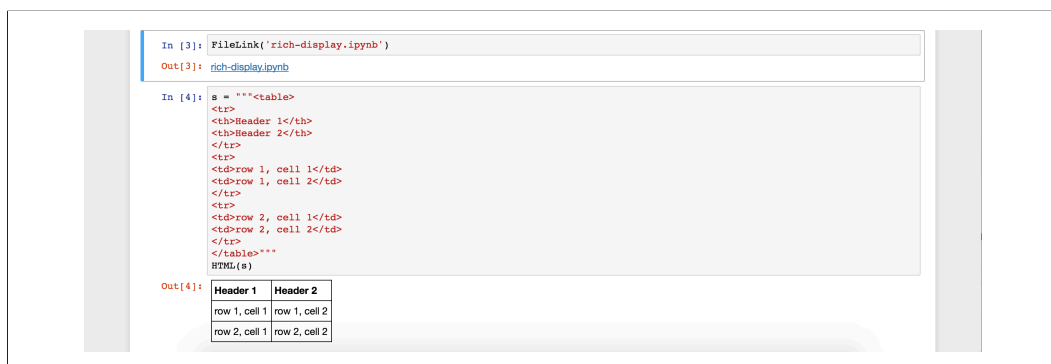


图 12.5 以可视化的样式显示表格

Pandas 的 DataFrame 对象也支持可视化显示，如图 12.6 所示。

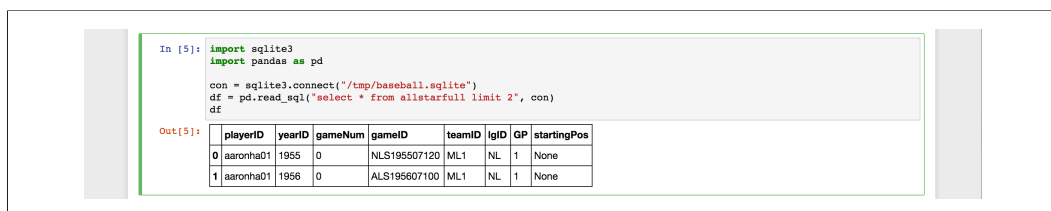


图 12.6 DataFrame 对象也支持可视化显示

能用这样的方式显示出来是因为定义了 `_repr_html_` 方法：

```
In : df._repr_html_()
```

```
Out: u'<div>\n<table border="1" class="dataframe">\n  <thead>\n    <tr style="text-align: right;">\n      <th></th>\n      <th>playerID</th>\n      <th>yearID</th>\n      <th>gameNum</th>\n      <th>gameID</th>\n      <th>teamID</th>\n      <th>lgID</th>\n      <th>GP</th>\n      <th>startingPos</th>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n      <th>0</th>\n      <td>aaronha01</td>\n      <td>1955</td>\n      <td>0</td>\n      <td>NLS195507120</td>\n      <td>ML1</td>\n      <td>NL</td>\n    </tr>\n    <tr>\n      <th>1</th>\n      <td>aaronha01</td>\n      <td>1956</td>\n      <td>0</td>\n      <td>ALS195607100</td>\n      <td>ML1</td>\n      <td>NL</td>\n    </tr>\n  </tbody>\n</table>\n</div>'
```

其实对任意对象添加这个方法都能自定义显示的方式:

```
class DictTable(dict):
    def _repr_html_(self):
        html = ['<table>']
        for key, value in self.iteritems():
            html.append("<tr><td>{0}</td><td>{1}</td></tr>".format(key, value))
        html.append("</table>")
        return ''.join(html)

t = DictTable(a=1, b=2, c=4)
display_html(t)
```

自定义 JavaScript 和 CSS 样式

Jupyter Notebook 提供了定制额外 JavaScript 代码和 CSS 样式的功能, 代码都存在于 `~/jupyter/custom/` 目录下。我们先看自定义的样式文件 `custom.css`, 它替换默认的 Notebook 的 Logo, 隐藏一些不常用或者危险操作的选项:

```
#menubar .navbar {
    visibility: hidden;
}

.btn-group {
    visibility: hidden;
}

#cell_type {
    visibility: hidden;
}

a[title="dashboard"] {
    display: none;
}
```

```
#ipython_notebook {
  background: url("/custom/dongxi.jpg") no-repeat scroll 0 0 / 100% auto rgba(0, 0,
    0, 0);
  height: 54px;
  width: 120px;
  border-radius: 3px;
}

.container.border-box-sizing.toolbar {
  visibility: hidden;
}

#run_int {
  visibility: visible;
  margin-left: -222px;
}
```

需要注意，JN 4.0 之后自定义的静态文件访问的地址不再是/static/custom，而是/custom。

看一下效果，如图 12.7 所示。

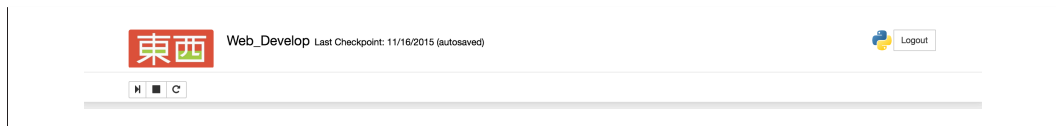


图 12.7 效果图

JN 的在线编辑器使用了开源的 CodeMirror，它自带支持 Emacs/VIM 的键盘绑定。现在我们定制使用 Emacs，修改 custom.js 的内容：

```
require(["base/js/events", "base/js/namespace", "notebook/js/cell",
  "codemirror/lib/codemirror", "codemirror/keymap/emacs"],
function(events, Jupyter, Cell, CodeMirror) {
  events.on('notebook_loaded.Notebook', function(){
    var load_ipython_extension = function() {
      var extraKeys = CodeMirror.keyMap.emacs;
      Cell.Cell.options_default.cm_config.extraKeys = extraKeys;
      Cell.Cell.options_default.cm_config.lineWrapping = true;

      var cells = Jupyter.notebook.get_cells();
      var numCells = cells.length;
      for (var i = 0; i < numCells; i++) {
        var theseExtraKeys = cells[i].code_mirror.getOption('extraKeys
          ');
        for (var k in extraKeys) {
```

```

        theseExtraKeys[k] = extraKeys[k];
    }
    cells[i].code_mirror.setOption('extraKeys', theseExtraKeys);
    cells[i].code_mirror.setOption('lineWrapping', true);
}
};

return {
    load_ipython_extension: load_ipython_extension,
};
});
});

```

使用 nbextension 扩展 Notebook

IPython 有扩展系统，比如之前提到的内置的 `autoreload`、`storemagic`，以及外部需要额外安装的 `cythonmagic` 等。JN 也有自己的扩展系统，叫作 `nbextension`。通过扩展都会存放在 `~/.jupyter/nbextensions` 目录下。最知名的扩展项目就是 `IPython-notebook-extensions`。我们先下载它：

```
> git clone https://github.com/ipython-contrib/IPython-notebook-extensions
```

拷贝 `nbextensions` 目录到 Jupyter 配置目录下：

```

> cp -rp IPython-notebook-extensions/nbextensions/ ~/.jupyter
> cp -rp IPython-notebook-extensions/nbextensions/config/ ~/.jupyter/nbextensions
> mkdir ~/.jupyter/extensions
> cp -rp IPython-notebook-extensions/extensions/nbextensions.py ~/.jupyter/extensions

```

在 `custom.js` 中添加希望自动加载的插件：

```
Jupyter.load_extensions('usability/chrome_clipboard');
```

给 `jupyter_notebook_config.py` 添加如下配置：

```

import os
import sys

from jupyter_core.paths import jupyter_config_dir, jupyter_data_dir

sys.path.append(os.path.join(jupyter_data_dir(), 'extensions'))

c = get_config()
...

```



```
c.NotebookApp.extra_template_paths = [os.path.join(jupyter_data_dir(), 'templates')]
c.NotebookApp.server_extensions = ['nbextensions']
```

启动 Notebook 之后可以访问 “https://127.0.0.1:5000/nbextensions” 配置插件。

打开 Notebook 页面就可以在浏览器终端可以看到如下的输出：

```
load_extensions ["jupyter-js-widgets/extension", "usability/chrome-clipboard/main"]
utils.js:36
Loading extension: usability/chrome-clipboard/main
utils.js:36
Loading extension: jupyter-js-widgets/extension
extension.js:88
loaded widgets
```

我们就可以使用这些扩展提供的功能了。扩展的功能和使用方法详见项目的 Wiki 页面。

在 Notebook 上使用并行计算

可以直接在 Notebook 里使用并行计算功能，但是需要先安装 `ipyparallel`，然后启动集群插件：

```
> ipcluster nbextension enable
```

先创建一个集群用的配置：

```
> ipython profile create mycluster --parallel
```

访问 “https://localhost:5000/tree#ipyclusters” 或者切换到 “IPython Clusters” 这个 Tab 上就可以启动集群，而不需要用命令行管理了，单击 `mycluster` 后面显示为 “Start” 的按钮，集群就会启动。如果在 # of engines 的输入框中未指定启动的进程数，会默认启动与逻辑 CPU 个数相同数目的进程，如图 12.8 所示。

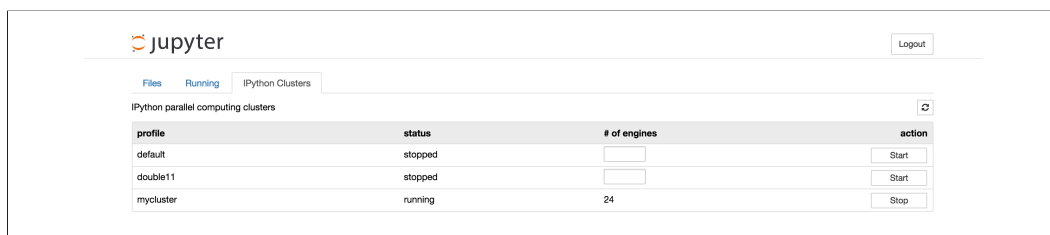


图 12.8 默认启动与逻辑 CPU 个数相同数目的进程

现在就可以在终端或者 Notebook 上连接这个集群了：

```
import ipyparallel as ipp
c = ipp.Client(profile='mycluster')
```

调试和 Debug 工具

之前介绍了 Sentry 帮我们发现应用中的问题,但是这还远远不够,我们还需要掌握如下内容:

- 了解 Linux 服务器的相关情况,如 CPU、内存、负载、网络、硬盘剩余资源等。
- 掌握性能测试工具。
- 掌握分析 Python 程序性能瓶颈的工具。

了解 Linux 服务器运行情况

free

在 Linux 下,使用 free 命令获取当前内存的使用情况:

```
> free -h
              total        used        free      shared    buffers     cached
Mem:           94G         92G         2.5G         842M          6.1M          73G
-/+ buffers/cache:      18G          76G
Swap:          3.0G           0B          3.0G
```

还有一个常见的参数是-m,表示以 MB 为单位。不过现在硬件越来越便宜,内存都以 GB 论,建议使用-h 让结果可读。上述结果中纵轴主要是 Mem 和 Swap,横轴表示各类型的值,如表 12.3 所示。

表 12.3 横轴值的类型与含义

类 型	含 义
total	内存总量
used	已经使用的内存量
free	还没有使用的内存量
shared	进程共享的内存量
buffers/cached	磁盘缓存的大小

纵轴上还有一行“-/+ buffers/cache”,其中“- buffers/cache”表示已用内存,它的值是 used - buffers - cached 的结果,也就是 92G - 6.1M - 73G 约等于 18 GB;“- buffers/cache”表示可用内存,它的值是 free + buffers + cached 的结果,也就是 2.5G + 6.1M + 73G 约等于 76 GB。

Swap 是交换空间,也就是虚拟内存,当物理内存不够的时候可以牺牲一些性能,不至于内存溢出而让程序停止运行。当内存比较小的时候,交换空间通常是推荐的做法。

如果经常发现可用内存不到全部内存的 30% 或者 Swap 的 used 的值不为 0，说明需要考虑增加物理内存了。

uptime

如果主观上感觉系统运行缓慢，首先可以通过 uptime 命令获知当前系统的平均负载：

```
> uptime
18:51:20 up 251 days, 6:26, 6 users, load average: 4.74, 4.30, 4.21
```

uptime 命令的输出：4.74、4.30、4.21，分别表示最近 1 分钟、最近 5 分钟和最近 15 分钟内进程队列中的平均进程数，进程数越多表示进程需要等待的时间越长，系统越繁忙。如果这三个值都经常超过逻辑 CPU 的个数，就说明 CPU 很繁忙。获得逻辑 CPU 个数的方式如下：

```
> grep processor /proc/cpuinfo|wc -l
24
```

可见上面的系统还是运行良好的。但是我们不一定非得等负载达到逻辑 CPU 的个数时再来解决，通常在当系统负荷持续大于 $0.7 \times$ 逻辑 CPU 的个数时，就应该开始研究问题出在哪里，想办法优化或者更新硬件了。



通过 w 和 top 等命令也可以输出 uptime 的内容。

htop

htop 是一款运行于 Linux 系统上的监控与进程管理软件，用于取代 UNIX 下传统的 top。top 只提供最消耗资源的进程列表，htop 提供所有进程的列表，并且使用颜色标识出处理器、Swap 和内存状态。htop 还提供方便光标控制的界面来杀死进程、过滤进程、搜索进程、根据指标排序等功能。

可以通过如下命令安装它：

```
> sudo apt-get install htop
```

在 htop 的默认设置中，提供如表 12.4 所示类型的输出。

默认情况下，这些输出是按照进程占用的 CPU 使用率从高到低排序的，也可以通过 MEM%、RES、USER 等指标排序。

表 12.4 Flask-DebugToolbar 内置的面板及功能

输出类型	含 义
PID	进程 ID
USER	运行此进程的用户
PRI	进程的优先级
NI	进程的优先级别值，默认的为 0
VIRT	进程占用的虚拟内存值
RES	进程占用的物理内存值
SHR	进程占用的共享内存值
S	进程的运行状况。R 表示正在运行；S 表示休眠，等待唤醒；Z 表示僵死状态
CPU%	该进程占用的 CPU 使用率
MEM%	该进程占用的物理内存和总内存的百分比
TIME+	该进程启动后占用的总的 CPU 时间
COMMAND	进程命令名称

dstat

dstat 是一个用 Python 语言实现的多功能系统资源统计生成工具，可以取代 vmstat、iostat、netstat 和 ifstat 等工具，它可以动态收集网络、硬盘、CPU 等资源情况。dstat 可以让你实时地看到所有系统资源，并以不同颜色显示出来。我们能通过观察一定时间段内资源的使用情况来分析问题。

我们先安装它：

```
> sudo apt-get install dstat
```

dstat 支持的参数非常多，我们分开演示：

```
> dstat -cdng
----total-cpu-usage---- -dsk/total- -net/total- ---paging--
usr sys idl wai hiq siq| read writ| recv send| in out
22  4  72  1  0  1| 721k 1757k|  0   0 |  0   0
22  3  74  0  0  1|  0  650k| 28M 9460k|  0   0
21  6  72  0  0  1|  0   48k| 27M 2270k|  0   0
22  8  68  0  0  1|  0  604k| 28M 9555k|  0   0
25  9  65  0  0  1| 72k   0 | 30M 2244k|  0   0
```

- -c 表示显示 CPU 占用信息，usr/sys/idl 分别表示用户进程执行时间/系统进程执行时间/空闲时间所占用的 CPU。如果 usr 长期超过 50，表示程序需要优化；如果 sys 比

较高，且 `idl` 经常小于 50，通常说明 CPU 负荷很重，应该考虑增加物理 CPU 或者更换服务器了。`wai` 表示在等待 I/O 设备（例如磁盘或者网络）的响应，如果 `wai` 长期处于一个比较大的值，说明 IO 等待比较严重，需要进一步研究造成等待的原因。

- `-d` 表示磁盘的读写操作。这一栏显示磁盘的读、写总数。
- `-n` 表示网络设备发送和接受的数据。这一栏显示网络收、发的数据总数。
- `-g` 表示分页活动。一个较大的分页表明系统正在使用大量的交换空间，或者说内存非常分散，最好的结果就是都输出 0。

`dstat` 还支持其他参数：

```
> dstat -lym
---load-avg--- ---system-- -----memory-usage-----
 1m  5m 15m | int   csw | used  buff  cach  free
5.18 4.43 4.31|5168   65k|19.8G 6224k 68.5G 6397M
5.32 4.47 4.33| 63k  107k|19.8G 6224k 68.5G 6392M
5.32 4.47 4.33| 61k  101k|19.8G 6224k 68.5G 6382M
5.32 4.47 4.33| 63k  106k|19.8G 6224k 68.5G 6389M
5.32 4.47 4.33| 62k  105k|19.8G 6224k 68.5G 6386M
```

其中 `-l` 和 `uptime` 命令、`-m` 和 `free` 命令的输出内容源是一样，只是添加了定期刷新的特性。`-y` 是系统统计，包含中断（`int`）和上下文切换（`csw`），通常值越大就表示有越多的进程造成了拥塞。

事实上，`dstat` 的强大不只是因为聚合了多种工具的系统检测结果，还因为它能通过附带插件而实现一些高级用法，如找出占用资源最高的进程和用户：

```
> dstat --proc-count --top-cpu --top-mem --top-io
proc -most-expensive- --most-expensive- ----most-expensive----
total|  cpu process   |  memory process |    i/o process
675|migration/14 4.2|tmux      1169M|redis-serve1090B  0
675|maelstrom-wor4.2|tmux      1169M|redis-serve2658B  0
675|maelstrom-wor4.1|tmux      1169M|redis-serve2660B  0
675|maelstrom-wor4.2|tmux      1169M|redis-serve2650B  0
675|maelstrom-wor4.1|tmux      1169M|redis-serve2659B  0
```

插件都存放在 `/usr/share/dstat` 目录下，可以参考它们的实现，并扩展更多的工具。

Glances

Glances 是一个基于 `psutil` 的跨平台的系统监控工具。它使用 Python 实现了一个比 `htop` 功能更齐全的监控工具。它支持 Docker 容器启动，支持 C/S 模型、还支持 SNMP 协议通信。

我们先安装它：

```
> pip install glances
```

如果你安装了 Bottle 这个 Web 框架，还能访问和终端显示几乎一样的 Glances 的 Web 页面，如图 12.9 所示。

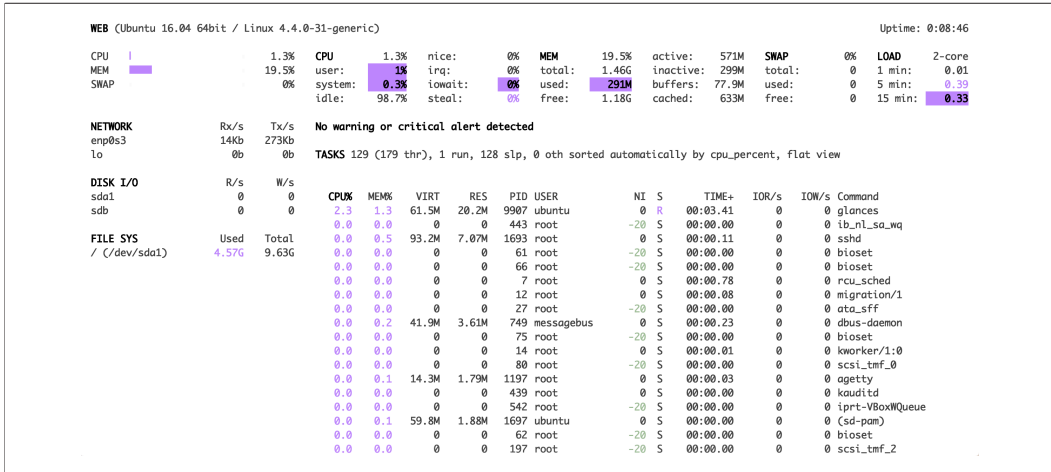


图 12.9 Glances 的 Web 页面

Glances 还支持把采集的数据导入到其他服务，如 InfluxDB、StatsD、ElasticSearch、RabbitMQ 等。

iftop

iftop 是类似于 top 命令的实时流量监控工具：

```
> sudo apt-get install iftop
```

iftop 展示了不同地址使用流量的情况。

sysdig

sysdig 是新兴的 Linux 服务器监控和故障排除工具。

```
> sudo apt-get install sysdig -y
```



在 Docker 环境中不能使用 sysdig。

在开始介绍 sysdig 之前，先了解如下几个命令。

- strace: 追踪某个进程产生和接收的系统调用。
- tcpdump: 监控原始网络数据包。
- lsof: 追踪打开的文件。

这三个命令之前是系统管理员必备的工具集的一部分。而 sysdig 其实相当于 strace + tcpdump + htop + iftop + lsof，它不仅能分析 Linux 系统的现场状态，还能将该状态保存为转储文件以供离线检查。

公众号文章《超级系统工具 Sysdig，比 strace、tcpdump、lsof 加起来还强大》(<http://bit.ly/29JJe65>) 介绍了包含网络、应用、容器、硬盘 I/O、进程和 CPU 使用率、安全等多个方面的用法，本书就不展开了。另外，官网的备忘单 (<http://bit.ly/29S1Dsp>) 一定要读读，里面对比了 sysdig 和其他工具命令的使用方法，熟悉这些方法可满足日常工作需要。

nethogs

nethogs 用来查看进程占用带宽情况：

```
➤ sudo apt-get install nethogs
```

使用起来也很简单：

```
➤ sudo nethogs eth0
```

使用 nethogs 很容易就找到流量占用最大的进程。

iotop

iotop 和 top 命令类似，只不过它是针对硬盘 I/O 进程的：

```
➤ sudo apt-get install iotop
```

当发现系统平均负载比较高、dstat 的 wai 字段值比较大时，可以使用 iotop 找出是哪些进程出了问题。

iptraf

iptraf 是一个网络流量监控工具：

```
➤ sudo apt-get install iptraf
```

iptraf 可以获取不同网卡的网络状况，甚至能查看到不同协议的进出流量数据包数和字节数等细节。

性能测试

一个网站到底能够承受多大的用户访问量，经常是开发者和系统管理员最关心的问题。开发者对应用程序进行优化以及系统管理员对网站架构进行调整以后，需要验证这些改变带来了多大的收益，除了进行灰度发布（A/B test 就是一种灰度发布），常用的办法就是对网站进行服务器压力测试了。

boom

本节将介绍一个用 Python 实现的压测工具 boom（<https://github.com/tarekziade/boom>），它是 Apache Bench 的替代品。

```
> pip install boom
```

它的使用也很简单：

```
> boom http://localhost:9000 -c 10 -n 100
Server Software: Werkzeug/0.11.3 Python/2.7.6
Running GET http://127.0.0.1:9000
Host: localhost
Running 100 times per 10 workers.
[=====>] 100% Done
```

```
----- Results -----
Successful calls      100
Total time           0.8056 s
Average              0.0753 s
Fastest              0.0198 s
Slowest              0.1006 s
Amplitude             0.0807 s
RPS                  124
BSI                   Pretty good
```

```
----- Status codes -----
Code 200              100 times.
```

```
----- Legend -----
RPS: Request Per Second
BSI: Boom Speed Index
```


-c 表示模拟并发 10 个用户，一共 100 次请求。

tcpcopy

使用 boom 或者其他压测工具时都有一个问题，就是模拟的请求不够真实，因为这种模拟并不能考虑到实际环境中复杂的网络状况，只能使用同一个客户端，请求单一。这个时候可以使用由网易工程师开发的 tcpcopy 导入线上流量进行功能和压力测试，这样做的好处是测试数据接近真实水平，而且实施起来相对简单。tcpcopy 支持 HTTP、Memcached、Redis、MySQL 等协议。

它有如下优点：

- 实时。
- 效果真实。因为 tcpcopy 是一种 TCP 请求复制工具，它复制的是真实的请求。
- 低负载，基本不影响线上应用。
- 操作简单。
- 分布式。

tcpcopy 架构有三个角色。

1. online server：线上服务器。tcpcopy 运行在这上面，捕捉和复制线上请求给 target server。
2. target server：路由服务器。处理复制过来的请求，然后将响应信息返回给 assistant server。
3. assistant server：辅助服务器。intercept 运行在这上面，主要负责一些辅助的工作，如将回应包信息传回给 tcpcopy。

我们实验的架构如下：

角 色	IP
online server	192.168.0.174
target server	192.168.0.132
assistant server	192.168.0.130

首先在 target server 上设置默认路由：

```
> sudo route add -host 192.168.0.241 gw 192.168.0.130
```

192.168.0.241 是我们最后伪装的源地址。tcpcopy 既支持 IP，也可以使用地址段。

现在确保 target server 上跑着应用：

```
> python -m SimpleHTTPServer 9000
```

在 assistant server 上安装 intercept：

```
> sudo apt-get install libpcap-dev -y
> wget https://github.com/session-replay-tools/intercept/archive/1.0.0.tar.gz
> tar xzf 1.0.0.tar.gz
> cd intercept-1.0.0
> ./configure && sudo make install
```

启动 intercept：

```
> sudo /usr/local/intercept/sbin/intercept -i eth1 -F 'tcp and src port 9000'
```

最后在 online server 上安装 tcpcopy：

```
> sudo apt-get install libpcap-dev -y
> wget https://github.com/session-replay-tools/tcpcopy/archive/1.0.0.tar.gz
> tar xzf 1.0.0.tar.gz
> cd tcpcopy-1.0.0
> ./configure && sudo make install
```

为了保持同样的 Python 应用，我们启动 SimpleHTTPServer：

```
> sudo python -m SimpleHTTPServer 80
```

启动 tcpcopy：

```
> sudo /usr/local/tcpcopy/sbin/tcpcopy -l /var/log/tcpcopy.log -x 80-192.168.0.132:9000
-s 192.168.0.130 -c 192.168.0.241
```

上面启动命令表示监控本机 80 端口的流量，把流量转发到 192.168.0.132 的 9000 端口上。

现在请求 online server 上 Web 服务：

```
> http http://192.168.0.174
```

这时候可以在 target server 上收到了同样的请求了：

```
192.168.0.241 - - [03/May/2016 08:15:57] "GET / HTTP/1.1" 200 -
```

可以看到请求的源地址被改变了。

Python 程序性能分析

cProfile/pstats

profile 和 cProfile 都可以用来收集程序执行的调用链的顺序、每一步所花费的时间、调用的次数、执行的模块名字或者代码文件以及具体的行数等信息。它们的工作原理都一样，唯一的区别是 cProfile 模块是以 C 扩展的方式实现的，一般情况下应该使用 cProfile。

模拟一个性能问题的例子（profile_test.py）：

```
import time

def func1():
    sum = 0
    for i in range(100000):
        sum += i

def func2():
    time.sleep(10)

func1()
func2()
```

使用如下方式调用 cProfile：

```
> python -m cProfile -o profile.out chapter12/section3/profile_test.py
```

上述的命令把收集到的性能分析数据存在 profile.out 中，需要借用可视化的工具查看。我们看一下使用 pstats 的效果：

```
> python -c "import pstats; p=pstats.Stats('profile.out'); p.sort_stats('time').
    print_stats()"
```

```
Tue May 3 16:39:15 2016    profile.out
```

```
6 function calls in 10.012 seconds
```

```
Ordered by: internal time
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	10.003	10.003	10.003	10.003	{time.sleep}
1	0.006	0.006	0.008	0.008	chapter12/section3/profile_test.py:5(func1)
1	0.002	0.002	0.002	0.002	{range}

```

1    0.001    0.001    10.012    10.012 chapter12/section3/profile_test.py:2(<
    module>)
1    0.000    0.000    10.003    10.003 chapter12/section3/profile_test.py:11(
    func2)
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
    objects}

```

可以看出测试程序中最耗费时间的是 `time.sleep`，竟然花了 10.003 s。`sort_stats` 支持多种排序类型，如 `calls`、`cumulative`、`line`、`pcalls`。

之前介绍的 `Flask-DebugToolbar` 和 `ProfilerMiddleware` 显示的性能分析的内容都是基于 `cProfile` 和 `pstats` 的。

豆瓣工程师看到的豆瓣页面都被添加了额外的内容，比如页面响应完成后，黑导航上面会显示本次请求的响应时间，如图 12.10 所示。



图 12.10 豆瓣工程师看到的豆瓣页面

右侧有一个显示为黄色的，内容为 216 ms 的可点击的链接，如果页面响应时间小于 200 ms，就是绿色的，表示响应正常；超过 400 ms 就是红色，表示太慢了，该修了。点击后就能看到页面 DOM 中插入的类似的性能分析结果。这样，工程师在使用豆瓣过程中就可以关注到问题，并且能方便快速地了解性能瓶颈在哪里。



还可以使用 `gprof2dot.py` 和 `graphviz` 图形化显示调用关系。

ipdb

`ipdb` 是添加了 `IPython` 支持的交互调试器，它的优点有支持自动补全、语法高亮、更好的接口自省等。我们先安装它：

```
> pip install ipdb
```

`set_trace()` 是最常用的方法，如果在复杂程序中发现了问题，可以在代码中插入 `set_trace()` 函数，并运行程序。当执行到 `set_trace()` 函数时，就会暂停程序的执行并直接跳转到调试器中，这时候我们可以检查相关的上下文。当退出调试器时，调试器会自动恢复程序的执行。下面展示一个在 `Flask` 视图中添加调试器的例子（`app_with_ipdb.py`）：

```
import ipdb

@app.route('/')
def index():
    number = request.args.get('number')
    ipdb.set_trace() # set_trace()要放在有问题的代码之前
    result = 100 / number
    return 'Result is: {}'.format(result)
```

现在请求视图就可以在终端看到请求的处理被暂停了：

```
> python app_with_ipdb.py
* Running on http://0.0.0.0:9000/ (Press CTRL+C to quit)
> /home/ubuntu/web_develop/chapter12/section3/app_with_ipdb.py(12)index()
   10     number = request.args.get('number')
   11     ipdb.set_trace()
---> 12     result = 100 / number
      13     return 'Result is: {}'.format(result)
      14
```

```
ipdb> p number # 打印变量
None
```

其他常用的参数有 `q`（退出）、`args`（列出当前行的参数和变量值）、`j`（跳到对应行执行）、`c`（继续执行，直到遇到下一个断点）、`r`（继续执行，直到当前函数返回）、`n`（继续执行，直到当前函数的下一行或者返回）等。

line_profiler

`line_profiler` 是一个按行计时和记录执行频率的工具，能非常直观地看到哪一行有性能问题。我们先安装它：

```
> pip install line_profiler
```

最常用的是使用 `@profile` 装饰器，我们看一个例子（`line_test.py`）：

```
@profile
def fib(n):
    if n in (1, 2):
        return 1
    return fib(n - 1) + fib(n - 2)
```

```
print fib(5)
```

不用关心装饰器 `profile` 还没有被定义，`kernprof` 可以在脚本运行的时候注入它：

```
> kernprof -l -v chapter12/section3/line_test.py
```

```
5
```

```
Wrote profile results to line_test.py.lprof
```

```
Timer unit: 1e-06 s
```

```
Total time: 2.2e-05 s
```

```
File: chapter12/section3/line_test.py
```

```
Function: fib at line 2
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
2					@profile
3					def fib(n):
4	9	8	0.9	36.4	if n in (1, 2):
5	5	4	0.8	18.2	return 1
6	4	10	2.5	45.5	return fib(n - 1) + fib(n - 2)

`-l` 这个选项是告诉 `kernprof` 将 `@profile` 装饰器注入到你的脚本的内建环境里；`-v` 是告诉 `kernprof` 在脚本执行完之后立马显示计时信息；如果不指定 `-o` 参数，默认会把分析结果存在 `line_test.py.lprof` 中。格式兼容 `pstats` 模块，但是没有按行的效果：

```
> python -c "import pstats; p=pstats.Stats('line_test.py.prof'); p.sort_stats('calls').print_stats()"
```

memory_profiler

`memory_profiler` 是一个监控 Python 代码内存使用量的工具。我们先安装它：

```
> pip install memory_profiler
```

强烈推荐安装 `psutil` 这个库来提高 `memory_profiler` 的性能。

跟 `line_profiler` 类似，`memory_profiler` 需要用 `@profile` 装饰器来装饰你感兴趣的函数，就像这样（`mem_test.py`）：

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
```

```
my_func()
```

用以下的命令来查看 `my_func` 在运行时耗费的内存：

```
> python -m memory_profiler chapter12/section3/mem_test.py
Filename: chapter12/section3/mem_test.py
```

Line #	Mem usage	Increment	Line Contents
2	18.207 MiB	0.000 MiB	@profile
3			def my_func():
4	25.844 MiB	7.637 MiB	a = [1] * (10 ** 6)
5	178.434 MiB	152.590 MiB	b = [2] * (2 * 10 ** 7)
6	25.844 MiB	-152.590 MiB	del b
7	25.844 MiB	0.000 MiB	return a

按行显示使用的内存量，能很直观地看到增加/减少的内存量。

`Line_profiler` 和 `memory_profiler` 共有的特性是它们在 IPython 里都有快捷方式。你只需要在 IPython 里输入以下内容：

```
In : %load_ext memory_profiler
In : %load_ext line_profiler
```

就可以使用 `%lprun` 和 `%mprun` 这两个 Magic 函数了。在 IPython 中使用它们时不能出现 `@profile` 装饰器（因为这种模式下 `profile` 未定义），需要使用未添加 `@profile` 装饰器的函数：

```
In : from chapter12.section3.ipyn_test import my_func
In : %mprun -f my_func my_func()
In : %lprun -f my_func my_func()
```

性能调优实践

如果你感觉程序很慢，首先应该分析慢的原因，然后再寻求优化的方法。性能调优一般可以从以下几个方面来着手。

1. 优化程序算法本身。通常先使用上述分析工具度量 CPU 时间、内存使用、函数调用次数等方面数据，然后可以选择合适的数据结构，优化循环，去掉不必要的抽象，避免不必要的拷贝和计算，以空间换时间（以时间换空间），充分利用系统资源（如多线程、多进程、多核并行计算）等。这些方法比较初级，但能消除明显的编程细节引起的瓶颈。
2. 优化运行环境和资源。运行环境与资源包括各种软硬件平台，包含操作系统、数据库、CPU、内存、磁盘、网络等。这是最直接、最简单的调优方法，如购买更好的 CPU，使用 SSD 等。这种方法受制于预算和硬件本身纵向扩展的限制。

3. 优化系统架构。单台服务器的处理能力总是会达到上限的，常见的思路是把压力分散到多台机器上，从而使每台机器都能获得可接受的延迟或吞吐量。除此之外，应该使用更合适的存储和缓存策略，如在符合条件的场景下使用 NoSQL。

进阶篇：定制基于 IPython 的交互解释环境

之前在介绍 Flask 0.11 的命令行接口时，实现了在 Flask 应用中使用基于 IPython 的交互解释环境的例子。这种模式在日常开发中非常有意义：我们不再需要每次进入交互环境之后都要 import 一大堆模块，而是直接使用它们。本节将继续深入讲解定制 IPython，下面就基于文件托管服务的代码实现一个这样的环境（shell.py）。

首先通过 readline 模块添加记录历史记录的函数：

```
import atexit

def hook_readline_hist():
    try:
        import readline
    except ImportError:
        return

    # 指定一个存储历史记录的文件地址
    histfile = os.environ['HOME'] + '/.web_develop_history'
    readline.parse_and_bind('tab: complete')
    try:
        readline.read_history_file(histfile)
    except IOError:
        pass # 第一次使用时文件还不存在

def savehist():
    try:
        readline.write_history_file(histfile)
    except:
        print 'Unable to save Python command history'
    atexit.register(savehist) # 添加退出时保存历史记录的钩子
```

接着添加环境的 Banner 信息，这非常重要，添加合理的 Banner 信息能让开发者直观地分辨开发环境和正式环境：

```
def get_banner():
    from app import app
    color_tmpl = '\x1b[{}m{}\x1b[0m'
    return color_tmpl.format(
        *(32, 'Development shell, do whatever you want.')
```



```

    if app.debug else (
        35, 'Production shell, use it carefully!'))

```

“\x1b[COLORMTEXT\x1b[0m” 是 ANSI 转义码（<http://ascii-table.com/ansi-escape-sequences.php>）中提供的打印带颜色文本的方法，其中 30~37 是文本颜色，40~47 是背景颜色，32 表示绿色，35 表示品红色。

IPython 的设置如下：

```

from IPython.terminal.prompts import Prompts, Token
from IPython.terminal.ipapp import TerminalIPythonApp
from IPython.terminal.interactiveshell import TerminalInteractiveShell

```

```

class MyPrompt(Prompts):
    def in_prompt_tokens(self, cli=None):    # default
        return [
            (Token.Prompt, 'In <'),
            (Token.PromptNum, str(self.shell.execution_count)),
            (Token.Prompt, '>: '),
        ]

    def out_prompt_tokens(self):
        return [
            (Token.OutPrompt, 'Out<'),
            (Token.OutPromptNum, str(self.shell.execution_count)),
            (Token.OutPrompt, '>: '),
        ]

class MyIPythonApp(TerminalIPythonApp):
    def init_shell(self):
        self.shell = TerminalInteractiveShell(
            prompts_class=MyPrompt, highlighting_style='emacs',
            display_banner=False, profile_dir=self.profile_dir,
            ipython_dir=self.ipython_dir, banner1=get_banner(), banner2='')
        self.shell.configurables.append(self)

app = MyIPythonApp.instance()
app.initialize()
app.shell.user_ns.update(user_ns)
sys.exit(app.start())

```

TerminalInteractiveShell 接受一系列参数，其中 MyPrompt 类把提示符改成了如下格式：

In <1>:

highlighting_style 可以指定语法高亮的 Pygments 样式的名字，全部样式可以通过如下命令获得：

```
import pygments
list(pygments.styles.get_all_styles())
```



从 IPython 5.0 开始，TerminalInteractiveShell 将使用第三方的 prompt-toolkit 作为新的终端接口，它提供如下特性：

- 纯 Python 实现。
- 通过 Pygments 库，支持语法高亮。
- 支持 Emacs/VIM 的键盘绑定。
- 轻量，只依赖 Pygments、six 和 wcwidth 就支持了 2.6 到 3.5 的 Python 版本。

这个库的作者还写了另外一个 REPL 的实现 ptpython。如果你想构建跨平台、接口简单、易移植的交互命令行和终端应用，prompt-toolkit 将是最好的选择。

进阶篇：豆瓣东西的 Jupyter Notebook 实践

之前深入介绍了 IPython、Pandas 等工具，那么它们在 Web 开发中又能发挥怎样的作用呢？豆瓣东西是豆瓣的电商导购产品，为了刺激用户消费、提高营收，豆瓣东西会不间断地和一些重要节日做一些专题活动，而“双十一”这一天也不例外。先看一下 2014 年双十一专题页面（<https://dongxi.douban.com/special/2014/1111/>）的一部分，如图 12.11 所示。



图 12.11 2014 年双十一专题页面

我们把图 12.11 中发布框之外的部分称为“推荐位”。运营同事会定期根据阿里后台以及我们自己的数据分析后台在管理后台调整推荐位的内容，曝光更多有价值的商品，向天猫带去更多的流量。

这种操作后台有如下 4 个特征。

- 临时性：这种后台有时效性，在一段时间内要用，之后可能就不再使用，或者只是为了快速实现一个 Demo，而未来的需求很可能会发生比较大的变化。
- 增加管理后台负担：产品开发要响应的运营、产品需求非常多，每个产品线都有一个到多个不同目的的管理后台，在产品不断尝试新方向和试错的过程中，会产生大量的后台，经过一段时间后，有些后台由于提需求的同事离职而荒废，有些由于不再跟进而被忽略，久而久之产生很多不知道是用来做什么，不清楚还有没有人用的后台页面，项目越堆越大。
- 完成的起点高：很多需求，Web 开发者希望糙、快、猛地实现，但是如果把它放入管理后台，必然要遵循当前后台使用的技术和用法、需要具备一定的前端能力。
- 改动部署不方便。

这个时候 Jupyter Notebook、Nbviewer 和 Pandas 这 3 个工具就非常有用。

- 使用 Nbviewer 搭建纯静态 HTML 页面应用，运营可以直接访问生成的静态页面查看数据分析结果，还能使用如 Echarts 这样的工具让数据图表化。
- 使用 Pandas 从数据库或者文件中把结果可视化地呈现给需求方，懂技术的产品同事甚至可以直接在 Jupyter Notebook 页面使用 SQL 语句进行一些查询。
- 临时后台使用 Jupyter Notebook 实现。不需要发生错误后去看 Sentry 对应的异常页面，直接在线上调试修复即可。开发者有非常大的自由度，修复问题要及时得多，而且部署方便。

当你熟悉以上 3 种工具，搭好架子，只需要一些前端知识，花少得多的时间就可以快速完成需求。

完成这个后台的第一步是创建一个内核（Kernel）。之前只介绍了官方提供的 Python 2 以及 Python 3 这两个内核，实际上创建一个内核非常容易，一般选择创建新的内核是希望多种需求可以分开管理，互不影响。

首先创建用户级别的内核目录：

```
➤ mkdir ~/.ipython/kernels
```



系统级别的目录包含 `/usr/share/jupyter/kernels` 和 `/usr/local/share/jupyter/kernels`。

我们只需要创建一个子目录，子目录下有个叫作 `kernel.json` 的文件：

```
> mkdir ~/.ipython/kernels/double11
> cat ~/.ipython/kernels/double11/kernel.json
{
    "display_name": "Double11",
    "language": "python",
    "argv": [
        "python",
        "-m", "ipykernel",
        "--profile=double11",
        "-f", "{connection_file}"
    ],
    "env": {
    }
}
```

Jupyter Notebook 并不能在配置中指定使用哪个 IPython 的配置文件，通常是指定自定义内核来实现设置的定制。现在创建 `double11` 这个配置：

```
> ipython profile create double11
```

然后在 `~/.ipython/profile_double11/startup` 下添加启动文件。使用这种启动 JN 就自动加载的程序，目的就是添加安全配置以及把一些代码提前准备好，而不用把大量的代码都放在 Jupyter Notebook 文档里，每次还需要执行。我们创建两个文件，首先看“双十一”推荐位相关的文件 `00-double11.py`（文件名字开始的 00 表示优先级，数字越小，优先级越高）。程序分为三部分，第一部分是封装 Redis，原因是 Redis 不支持存储复杂对象，所以封装的作用是存储 cPickle 序列化之后的字符串，取出来的时候再反序列化：

```
from __future__ import unicode_literals

import cPickle as pickle

import redis

class RedisWrapper(object):
    def __init__(self):
        self.r = redis.StrictRedis(host='localhost', port=6379)

    def __getattr__(self, attr):
```

```

        if attr == 'r':
            return vars(self)['r']
        return pickle.loads(self.r.get(attr))

    def __setattr__(self, key, value):
        if key == 'r':
            vars(self)['r'] = value
        else:
            pickled_object = pickle.dumps(value)
            self.r.set(key, pickled_object)

r = RedisWrapper()

```

第二部分是抽象两个公用的按钮函数：

```

def get_btn(desc):
    return widgets.Button(description="提交{}修改".format(desc))

def hide_btn(btn):
    print('提交完成')
    btn.visible = False
    sleep(1.5)
    btn.visible = True

```

第三部分就是对应的推荐位模块的逻辑了。通过上面的图片可以看到，推荐位分为三块：三个图文主题（双11购物攻略）、三个商品、两个精选豆列，每个豆列5个商品。

图文主题的逻辑如下：

```

def set_review():
    '''修改图文'''
    def review_func(btn):
        _review_ids = [
            review_container.children[i].value for i in range(3)]
        r.review_ids = _review_ids
        hide_btn(btn)

    review_container = widgets.Box()
    btn = get_btn("图文")
    children = []
    for o in range(3):
        c = widgets.Text(description="第{}条图文ID".format(o + 1))
        c.value = str(r.review_ids[o])
        children.append(c)
    children.append(btn)
    review_container.children = children

```

```
btn.on_click(review_func)
display(review_container)
```

精选豆列逻辑如下：

```
def set_doulist():
    '''修改豆列推荐位'''
    def doulist_func(btn):
        doulists = []
        for i in range(2):
            doulist = []
            doulist_id = doulist_container.children[i].children[0].value
            doulist.append(doulist_id)
            _stories = []
            for p in range(1, 6):
                _stories.append(
                    doulist_container.children[i].children[p].value)
            doulist.append(_stories)
            doulists.append(doulist)
        r.promo_ids = doulists
        hide_btn(btn)

    doulist_container = widgets.Box()
    btn = get_btn("豆列推荐位")
    children = []
    for doulist_id, stories_ids in r.promo_ids:
        doulist = widgets.Box()
        _children = []
        c = widgets.Text(description="豆列ID")
        c.value = str(doulist_id)
        _children.append(c)
        for sid in stories_ids:
            c = widgets.Text()
            c.value = str(sid)
            _children.append(c)
        doulist.children = _children
        children.append(doulist)
    children.append(btn)
    doulist_container.children = children
    btn.on_click(doulist_func)
    display(doulist_container)
    doulist_container._dom_classes = ('doulist',)
```

通过在按钮的 on_click 事件上加回调函数的方式，实现点击按钮数据就写回到数据库的功能，以达到实时修改的目的。

由于设置商品用法和设置豆列的思路相同，这里就不展示代码了。

现在有个用户体验的问题，怎么让非技术部门的同事可以非常方便地使用这样的管理后台呢？在上面写逻辑的过程中，我使用了“`doulist_container_dom_classes = ('doulist',)`”，也就是给显示出来的 JN 组件的样式加了额外的 CSS 类名，现在添加一些样式：

```
> mkdir ~/.jupyter/custom/
> cp chapter12/section5/custom.css ~/.jupyter/custom/
```

启动 JN：

```
> jupyter notebook --port=9000 --ip=0.0.0.0
```

看一下页面效果（<http://localhost:9000/notebooks/chapter12/section5/double11.ipynb>），如图 12.12 所示。

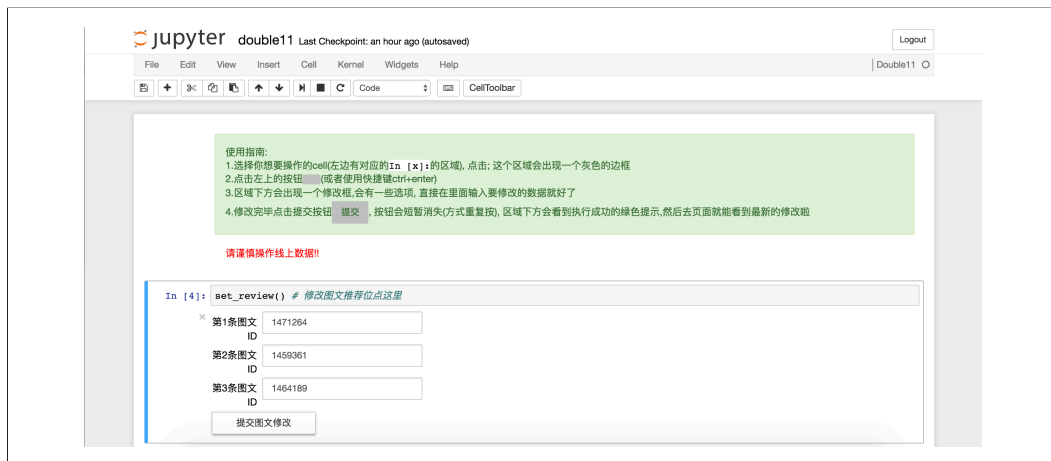


图 12.12 页面效果

其中第一个 Cell 是使用指南。它是一段 Markdown 类型的 Cell，内容是一段 HTML 代码：

```
<div class="alert alert-success guide">
<p>使用指南:</p>
<p>1.选择你想要操作的cell(左边有对应的`In [x]:`的区域)，点击；这个区域会出现一个
  灰色的边框</p>
<p>2.点击左上的按钮<button class='btn'><i class="icon-play"></i></button>(或者使用快
  捷键ctrl+enter)</p>
<p>3.区域下方会出现一个修改框,会有一些选项，直接在里面输入要修改的数据就好了</p>
<p>4.修改完毕点击提交按钮<button class="btn">提交</button>，按钮会短暂消失(防止重
  复按)，区域下方会看到执行成功的绿色提示,然后去页面就能看到最新的修改啦</p>
</div>
<div class="alert alert-error">请谨慎操作线上数据!!</div>
```

再看修改商品和豆列推荐位的效果，如图 12.13 和图 12.14 所示。



图 12.13 修改商品和豆列推荐位的效果 1



图 12.14 修改商品和豆列推荐位的效果 2

通过在 custom.css 中添加样式让后台展示的样式风格和专题页对应区域布局一致，运营人员就不会改错了。



如果想在本地运行这个例子，需要先向 Redis 中插入模拟数据：

➤ `python ~/web_develop/chapter12/section5/init_db.py`

第 13 章

Python 并发编程

现在拥有多核 CPU 的服务器随处可见，充分利用语言特性，使用并发方式编程也是 Web 开发者一项必须掌握的日常技能。好的并发程序远比单线程单进程的程序的运行效率高得多。本章我们通过抓取微信公众号文章内容并存入 MongoDB 数据库，来演示 Python 的并发技术。

编写爬虫（Crawler）程序算是不同级别 Python 工程师都可能涉及的工作之一，所谓爬虫就是让程序自动访问目标网站，解析页面，把需要的内容保存下来。写好爬虫其实原则只有一条，就是让你的抓取行为和用户访问网站的真实行为尽量一致。举些例子，用户不会 1 秒钟打开 10 个页面，也不会使用不符合浏览器格式的用户代理（User-Agent，UA），有些页面的 Referfer 和浏览器 Cookie 也很重要。为了保证爬虫持续稳定，甚至于还要模拟用户的正常行为，如发帖、签到、给别人点赞、参与讨论等。这些细节做好了，基本上什么网站都可以抓取成功。

为了让抓取顺利，我们要做好如下 4 件事：

1. 使用代理。一般的网站都有一些防爬虫的策略，如限制单位时间内页面的请求次数，基于来源 IP、UA 等请求信息判断用户访问是否正常。单机抓取整个网站内容，要想快速完成显然是不可能的，那么就需要分布式地使用多个服务器来抓取。如果公司没有专门的抓取服务器，可以从网上找一些代理服务器来使用。
2. 伪造 UA 字符串。每次请求都使用随机生成的 UA。
3. 选择解析 HTML 的方式。BeautifulSoup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。我们还需要给它配置一个解析器，解析器提供了相同的、非常人性化的接口，使用非常简便。常见的解析器包含如下几种：

- `html.parser`。使用 Python 标准库自带的解析器，文档容错能力强，但是速度上没有优势。
 - `lxml`。`lxml` 是一个 C 语言实现的 `libxml2` 和 `libxslt` 的 Python 绑定库，速度比其他的选择快得多，而且文档容错能力强。
 - `html5lib`。容错性最好，它的解析方式和其他解析器相比有所不同，它会以浏览器的方式解析文档。
4. 使用 Referfer。使用 Referfer 是一个好习惯，模仿一个从搜索引擎点击进来的请求，更不容易被封禁。

由于 BeautifulSoup 的易用性和 lxml 效率，笔者选择两者的组合来实现页面的解析。先安装它们。

```
> pip install beautifulsoup4 lxml
```

需要抓取的内容分三种：

- 抓取多个代理网站上发布的代理地址，把解析的代理地址存进数据库备用。将使用多线程完成。
- 抓取微信工作平台的搜索列表页。我们使用搜狗的微信搜索平台（<http://weixin.sogou.com>）。搜索支持两种类型，分别是微信公众号和微信文章，本章将抓取包含 Python 关键词的文章，也就是使用 `type=2`。将使用协程完成。
- 抓取微信文章内容，微信文章的域名是 <http://mp.weixin.qq.com>。将分别使用多进程、Future 和 `asyncio` 三种方式完成。

为了减少复杂度，随机生成 UA 的功能通过第三方库 `fake-useragent` 实现：

```
> pip install fake-useragent
```

生成一个 UA 字符串只需要如下代码即可：

```
In : from fake_useragent import UserAgent
In : ua = UserAgent()
In : ua.random
Out: u'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /41.0.2228.0 Safari/537.36'
In : ua.random
Out: u'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/35.0.1916.47 Safari/537.36'
```

我们使用 MongoEngine 管理数据模型。先定义基类：

```
from mongoengine import connect, StringField, DateTimeField, Document
```

```
from config import DB_HOST, DB_PORT, DATABASE_NAME

connect(DATABASE_NAME, host=DB_HOST, port=DB_PORT)

class BaseModel(Document):
    create_at = DateTimeField()

    meta = {'allow_inheritance': True,
            'abstract': True}
```

BaseModel 的 meta 属性将允许其被继承。

config.py 存放配置常量：

```
DB_HOST = 'localhost'
DB_PORT = 27017
DATABASE_NAME = 'chapter13'
```

为了让例子更简单易懂，本章的例子都使用过程式编程。

使用多线程

讨论 Python 程序的效率时，经常提到的弊端之一是全局解释器锁（Global Interpreter Lock, GIL），GIL 限制任何时间都仅有一个线程在执行。但是存在即合理，由于线程是轻量级的，并且相互之间易于通信，GIL 保护了所有全局的解释器和环境状态变量。如果不使用 GIL 会带来包括死锁、争用条件和高复杂性在内的各种问题。那么由于 GIL 的限制，使用多线程就完全没有意义了吗？其实不是的，原因如下：

1. Python 大部分是使用 C 语言编写的，一些标准库和第三方的模块也是用 C 编写的，而 C 语言代码是可以获取和释放 GIL 的，这在一定程度上缓解了 GIL 的问题。一些重要的、需要更高运行效率的模块还可以使用 C 语言编写，这样就可以利用更多的 CPU 资源。
2. 多线程适合解决由于网络、磁盘等资源造成的 I/O 阻塞问题，它利用了等待 I/O 请求完成被阻塞而导致的 CPU 空闲时间。计算密集型的工作不适合使用多线程完成，因为计算需要 CPU 资源，如果 CPU 繁忙的话，使用多线程并没有更多地利用 CPU 空闲时间，反而由于 CPU 需要额外调度多线程，以及线程切换的各种开销，多线程的运行效率比单线程还要慢。

抓取 HTTP 代理列表，主要是网络请求，几乎没有计算相关的工作，适合使用多线程。在开始使用多线程之前，先添加基本设置到 config.py 中：

```
import re

PROXY_SITES = [
    'http://xxx.com',
    ...
]

REFERER_LIST = [
    'http://www.google.com/',
    ...
]

获得 UA 和 Referer 以及请求的函数 (utils.py):

import random

import requests
from fake_useragent import UserAgent

from config import REFERER_LIST, TIMEOUT

def get_referer():
    return random.choice(REFERER_LIST)

def get_user_agent():
    ua = UserAgent()
    return ua.random
```



fake_useragent 在第一次启动的时候会将 <http://useragentstring.com> 网站上列出来的 UA 缓存到/tmp/fake_useragent.json, 这个过程可能需要代理才能完成, 可以使用项目源码目录下已经保存的文件:

```
> cp ~/web_develop/data/fake_useragent.json /tmp
```

添加代理的模型 Proxy:

```
class Proxy(BaseModel):
    address = StringField(unique=True) # 保证地址唯一

    meta = {'collection': 'proxy'} # 指定Model使用的集合(表)名
```

如果爬取非常多的代理网站, 是不是要针对每个网站都写一个解析 HTML 或者代理列表呢? 其实不用, 直接解析 HTML 页面中的代理地址就好了。不需要那么严格地匹配, 因为之后

我们还会验证这些地址，如果该代理地址不能使用，会自动把它从数据库中删掉的。匹配代理地址的正则表达式是：

```
PROXY_REGEX = re.compile('[0-9]+(?:\.[0-9]+){3}:\d{2,4}')
```

HTTP 请求函数如下：

```
def fetch(url, proxy=None):
    s = requests.Session()
    s.headers.update({'user-agent': get_user_agent()})

    proxies = None
    if proxy is not None:
        proxies = {
            'http': proxy,
        }
    return s.get(url, timeout=TIMEOUT, proxies=proxies)
```

本节未使用代理，`proxy` 参数为默认的 `None`。

保存代理的函数如下：

```
def save_proxies(url):
    try:
        r = fetch(url)
    except requests.exceptions.RequestException:
        return False
    addresses = re.findall(PROXY_REGEX, r.text)
    for address in addresses:
        proxy = Proxy(address=address)
        try:
            proxy.save()
        except NotUniqueError:
            pass
```

有可能多个代理网站上都列出了某个代理地址，所以在上面的实现中忽略了 `NotUniqueError` 这种异常。这里强调一点，应该指定捕获的异常的类型，不应该使用模糊的捕获，如下例：

```
try:
    l[0]
except:
    pass
```

应该使用：

```
try:
    l[0]
except IndexError:
```

```
pass
```

因为 `l[0]` 只可能产生 `IndexError` 这个异常。作为开发者，你应该明确地知道会出现哪些类型的异常，把它们都列出来。使用异常的基类可以指定更少的错误类型，比如上面用到的 `RequestException`，`requests` 的异常都以它为基类，在抓取的时候捕捉由 `requests` 发出的全部异常，不管它是 `Timeout` 还是 `ConnectionError`。但是使用 `Exception` 并不是一个好的习惯，虽然它是一般自定义异常的基类，但是可能会把一些非预期产生的错误忽略掉，除非你明确知道这样做的后果。

为了确保执行的前提环境一致，抓取前我们都会清空 `Proxy` 表：

```
def cleanup():
    Proxy.drop_collection()
```

代理有时效性，现在可用的代理可能过一会儿就不可用了。应该周期性替换成最新的代理。

如果不使用多线程，可以用一个 `for` 循环，挨个网站地抓取：

```
def non_thread():
    cleanup()
    for url in PROXY_SITES:
        save_proxies(url)
```

这种串行执行的效率很低：

```
In : from proxy_fetcher import non_thread
In : time non_thread()
CPU times: user 1.32 s, sys: 0 ns, total: 1.32 s
Wall time: 15.6 s
In : from models import Proxy
In : Proxy.objects.count()
Out: 376
```

看一下使用多线程的例子：

```
def use_thread():
    cleanup()
    threads = []
    for url in PROXY_SITES:
        t = threading.Thread(target=save_proxies, args=(url,))
        t.setDaemon(True)
        threads.append(t)
        t.start()

    for t in threads:
        t.join() # 等到线程都结束，再退出主程序
```

看一下效果，注意看花费的时间：

```
In : time use_thread()
CPU times: user 1.51 s, sys: 148 ms, total: 1.66 s
Wall time: 5.42 s
```

可以感受到速度提高了 3 倍，但是这个例子有一个不好的用法：一个站点就创建一个线程，如果站点很多就需要创建很多个线程，而创建和销毁线程是一个比较重的开销。可以考虑使用线程池，重用线程池中的线程：

```
from multiprocessing.dummy import Pool as ThreadPool

def use_thread_pool():
    cleanup()
    pool = ThreadPool(5)
    pool.map(save_proxies, PROXY_SITES)
    pool.close()
    pool.join()
```

看一下效果：

```
In : time use_thread_pool()
CPU times: user 1.41 s, sys: 93 ms, total: 1.51 s
Wall time: 5.25 s
```

可以看到，在限制了使用线程数的前提下，和上面不限制线程数的方式保持同等水平的运行效率。

`multiprocessing.dummy` 模块与 `multiprocessing` 模块的区别在于：前者是多线程，而后者是多进程，但是它们的接口都是一样的，可以很方便地将代码在多线程和多进程之间切换。在不熟悉并发编程的时候，可以先使用非并发的模式做出大部分功能，然后再迁移到多线程或者多进程的版本上，使用这种接口一致的方式就会让迁移成本很低。

之前的例子就是单纯地爬取入库，并没有返回抓取结果。在爬取结束时，我们还可以将抓取到的代理列表存放在某个变量中以备使用，这表示在多线程中需要共享数据。如果数据共享时可能被修改，就需要加锁来保护它，以确保同一时刻只能有一个线程访问这个数据。线程模块提供了许多同步原语，包括锁（Lock）、信号量（Semaphore）、条件变量（Condition）和事件（Event）。但是最好的做法是使用 `Queue` 模块。`Queue` 是线程安全的，使用它可以降低程序的复杂度，代码清晰、可读性更强。

先改成使用 `Queue` 的方式共享可修改数据：

```
import Queue

def save_proxies_with_queue(queue):
```

```

while True:
    url = queue.get()
    save_proxies(url)
    queue.task_done() # 向任务已经完成的队列发送一个信号

def use_thread_with_queue():
    cleanup()
    queue = Queue.Queue()

    for i in range(5):
        t = threading.Thread(target=save_proxies_with_queue, args=(queue,))
        t.setDaemon(True)
        t.start()

    for url in PROXY_SITES:
        queue.put(url)

    queue.join()

```

一定要调用 `task_done`，只有这样，在最后 `join` 方法时才能知道队列中的所有任务是不是都执行完了。

看一下效果：

```

In : from proxy_fetcher_with_queue import use_thread_with_queue
In : time use_thread_with_queue()
CPU times: user 1.46 s, sys: 0 ns, total: 1.46 s
Wall time: 6.21 s

```

使用 `Queue` 效率虽然差一些，但是扩展性很好。现在增加复杂度，抓取之后要把结果保留下来，放进一个列表中。我们使用多个队列，把抓取到的结果放置到另一个队列中。

首先改造 `save_proxies`，让它返回当前线程获得的代理地址：

```

def save_proxies(url):
    proxies = []
    try:
        r = fetch(url)
    except requests.exceptions.RequestException:
        return False
    addresses = re.findall(PROXY_REGEX, r.text)
    for address in addresses:
        proxy = Proxy(address=address)
        try:
            proxy.save()
        except NotUniqueError:
            pass

```



```
        else:
            proxies.append(address)
    return proxies
```

添加使用队列的 `save_proxies_with_queue2` 函数：

```
def save_proxies_with_queue2(in_queue, out_queue):
    while True:
        url = in_queue.get()
        rs = save_proxies(url)
        out_queue.put(rs)
        in_queue.task_done()
```

也就是从 `in_queue` 获取要执行的网站地址，把结果放进 `out_queue` 队列中。第二步是把放入 `out_queue` 队列中的结果存入一个全局的列表：

```
def append_result(out_queue, result):
    while True:
        rs = out_queue.get()
        if rs:
            result.extend(rs)
        out_queue.task_done()
```

`result` 就是一个全局变量，由于 Python 参数是引用传递，可以直接修改它。

现在基于 `use_thread_with_queue`，将其修改为函数 `use_thread_with_queue2`：

```
def use_thread_with_queue2():
    cleanup()
    in_queue = Queue.Queue()
    out_queue = Queue.Queue()

    for i in range(5):
        t = threading.Thread(target=save_proxies_with_queue2,
                              args=(in_queue, out_queue))
        t.setDaemon(True)
        t.start()

    for url in PROXY_SITES:
        in_queue.put(url)

    result = []

    for i in range(5):
        t = threading.Thread(target=append_result,
                              args=(out_queue, result))
        t.setDaemon(True)
```

```
t.start()

in_queue.join()
out_queue.join()

print len(result)
```

看一下运行效果：

```
In : time use_thread_with_queue2()
1806
CPU times: user 1.37 s, sys: 133 ms, total: 1.5 s
Wall time: 2.43 s
```

可以看到最后执行 `len(result)` 的结果也是 376。



本节通过 `time` 计算的任务执行时间会受限于网络等客观原因，笔者选择了几次运行时间中最短的那一次结果，仅供参考。

使用 Gevent

高并发编程时，采用多线程（或进程）是一种不可取的解决方案，因为线程（或进程）本质上都是操作系统的资源。每个线程都是需要额外占用内存的，由于线程的调度由操作系统完成，调度器会因为时间片用完等原因强制夺取某个线程的控制权，开发者还需要考虑加锁、使用队列等操作，这些都容易造成高并发情况下的性能瓶颈。

协程是用户空间线程，复杂的逻辑和异步都封装在底层，开发者还是在使用同步的方式编程，但这种协作式的任务调度可以让用户自己控制使用 CPU 的时间，除非自己放弃，否则不会被其他协程抢夺到控制权。

Python 2 通过 `yield` 提供了对协程的基本支持，但是功能很有限。而第三方的 `Gevent` 为 Python 提供了比较完善的协程支持。`Gevent` 是一个基于微线程库 `Greenlet` 的并发框架，虽然与直接使用 `Greenlet`、`Eventlet` 相比性能略低，但是它提供了和线程模型编程相仿的接口，而且提供 `Monkey Patch` 方法，可以在运行时动态修改标准库里大部分的阻塞式系统调用，如 `socket`、`threading` 和 `select` 等模块，让其变为协作式运行。

通过如下例子看一下 `Gevent` 的上下文切换（`gevent_spawn.py`）：

```
import gevent
```

```
def a():
```

```
    print 'Start a'
    gevent.sleep(1)
    print 'End a'

def b():
    print 'Start b'
    gevent.sleep(2)
    print 'End b'

gevent.joinall([
    gevent.spawn(a),
    gevent.spawn(b),
])
```

执行的结果如下：

```
Start a
Start b
End a
End b
```

在 Gevent 里面，上下文切换是通过 yield 来完成的，执行 `gevent.sleep` 会触发上下文切换，这个切换实际上交由程序来控制。

在开始爬取搜索页的工作之前，我们先使用 Gevent 过滤出可用的代理列表。为什么这么做呢？代理网站上发布的代理其实大部分是不可用的，为了不浪费我们的抓取资源，可以先过滤掉大部分不可用的代理地址（`remove_unavailable_proxy.py`）：

```
from gevent.pool import Pool
from requests.exceptions import RequestException

from utils import fetch
from models import Proxy

pool = Pool(10)

def check_proxy(p):
    try:
        fetch('http://baidu.com', proxy=p['address'])
    except RequestException:
        p.delete()

pool.map(check_proxy, Proxy.objects.all())
```

如果代理不能访问百度就证明其不可用，然后将其从数据库中删除。

看一下过滤后的代理数量：

```
In : Proxy.objects.count()
Out: 123
```

代理数量减少到原来的 1/3，可见免费代理很不稳定。除了寻找更多的代理网站，还可以使用自己的代理服务器，保证抓取的效率和稳定。

然后给 Proxy 类添加获得随机文档的类方法：

```
@classmethod
def get_random_proxy(cls):
    proxy = cls.objects.aggregate({'$sample': {'size': 1}}).next()
    return proxy
```

每次执行 Proxy.get_random() 都可以随机获得一个 Proxy 文档，尽量每次都使用不同的代理来抓取，杜绝长时间使用单 IP 高频抓取。

文章都有发布者，为了未来检索起来方便，把发布者用独立的模型存放：

```
class Publisher(BaseModel):
    display_name = StringField(max_length=50, required=True)

    meta = {'collection': 'publisher'}

    @classmethod
    def get_or_create(cls, display_name):
        try:
            return cls.objects.get(display_name=display_name)
        except DoesNotExist:
            publisher = cls(display_name=display_name)
            publisher.save()
            return publisher
```

笔者特意添加了一个叫作 get_or_create 的类方法，保证通过发布者的名字就能获得对应的 Publisher 文档。

接着定义文章的模型：

```
class Article(BaseModel):
    title = StringField(max_length=120, required=True)
    img_url = StringField()
    url = StringField(required=True)
    summary = StringField()
    publisher = ReferenceField(Publisher, required=True)
    create_at = DateTimeField()
```

```

meta = {
    'collection': 'article',
    'indexes': [
        '-create_at',
        {'fields': ['title', 'summary', 'publisher'],
         'unique': True}
    ],
    'ordering': ['-create_at']
}

```

搜索页抓取到的内容还不全，13.2 节会把其余字段填充进来。为了提高查询效率，添加了两个索引，一个是按照创建时间降序，另外一个 `title + summary + publisher` 的复合索引，保证不会插入三者结果值都相同的文章，索引不使用 `url` 字段，是因为同一篇文章每次获得文章地址的结果是不一样的。`Article` 对象默认以时间降序排序。

保存搜索结果的函数如下：

```

from mongengine import DoesNotExist
from gevent import sleep, GreenletExit

def save_search_result(page, queue, retry=0):
    proxy = Proxy.get_random()['address']
    url = SEARCH_URL.format(SEARCH_TEXT, page)

    try:
        r = fetch(url, proxy=proxy)
    except (Timeout, ConnectionError):
        sleep(0.1)
        retry += 1
        if retry > 5:
            queue.put(page)
            raise GreenletExit()
    try:
        p = Proxy.objects.get(address=proxy)
        if p:
            p.delete()
    except DoesNotExist:
        pass

    return save_search_result(page, queue, retry)
soup = BeautifulSoup(r.text, 'lxml')
results = soup.find(class_='results')
if results is None:
    # 此代理已经被封，换其他的代理

```

```

        sleep(0.1)
        retry += 1
        if retry > 5:
            queue.put(page)
            print 'retry too much!'
            raise GreenletExit()
        return save_search_result(page, queue, retry)
articles = results.find_all(
    'div', lambda x: 'wx-rb' in x)
for article in articles:
    save_article(article)

```

如果执行 fetch 遇到 Timeout 和 ConnectionError，一般说明代理有问题，异常处理中会找到这个 Proxy 对象，从 MongoDB 中把它删掉。

其次是 fetch 的请求如果失败，会 sleep 一段时间，这里设置为的 0.1 只是为了演示，实际环境中可适当加大。如果重试次数超过 5 次就会抛出 GreenletExit 异常，保证不会因为大量代理异常造成协程不能结束的情况。参数包含 queue 的唯一原因，就是在重试失败之前可以把任务再放回队列，保证任务不丢失。

可以看到使用 BeautifulSoup 的 find 和 find_all 这两种方法就能实现找到对应元素属性和文本的结果。唯一要注意的是，由于“class”在 Python 中的作用是声明类，所以使用 CSS 类的条件时需要使用“class_”替代“class”关键词。

下面这个函数解析单篇文章需要的字段内容并保存到 MongoDB 中：

```

from pymongo.errors import InvalidBSON
from mongoengine import NotUniqueError

```

```

def save_article(article_):
    img_url = article_.find(class_='img_box2').find(
        'img').attrs['src'].split('url=')[1]
    text_box = article_.find(class_='txt-box')
    title = text_box.find('h4').find('a').text
    article_url = text_box.find('h4').find('a').attrs['href']
    summary = text_box.find('p').text
    create_at = datetime.fromtimestamp(float(text_box.find(
        class_='s-p').attrs['t']))
    publisher_name = text_box.find(class_='s-p').find('a').attrs['title']

    article = Article(img_url=img_url, title=title, url=article_url,
                      summary=summary, create_at=create_at,
                      publisher=Publisher.get_or_create(publisher_name))

    try:
        article.save()

```

```
except (NotUniqueError, InvalidBSON):  
    pass
```

使用 Gevent 的协程池和队列实现并发抓取：

```
from gevent import monkey  
from gevent.queue import Queue, Empty  
from gevent.pool import Pool  
monkey.patch_all()
```

```
SEARCH_URL = 'http://weixin.sogou.com/weixin?query={}&type=2&page={}'  
SEARCH_TEXT = 'Python'
```

```
def use_gevent_with_queue():  
    queue = Queue()  
    pool = Pool(5)  
  
    for p in range(1, 7):  
        queue.put(p)  
  
    while pool.free_count():  
        sleep(0.1)  
        pool.spawn(save_search_result_with_queue, queue)  
  
    pool.join()
```

我们创建一个包含 5 个线程的池，搜索关键词为 Python 的文章。默认先把第 1 到第 6 页作为地址初始任务放进队列。队列的执行者 `save_search_result_with_queue` 函数如下：

```
def save_search_result_with_queue(queue):  
    while 1:  
        try:  
            p = queue.get(timeout=0)  
        except Empty:  
            break  
  
        save_search_result(p, queue)  
    print 'stopping crawler...'
```

如果队列为空，当前的协程就会结束。这里笔者有个经验，为了实现更好的扩展性，放入队列的是那些有变化的内容，也就是只放入页面数字，在 `save_search_result` 函数中，通过页数和搜索文本拼出 url，而不是把整个 url 拼好了再放入队列，这样除了让序列化工作更快以外，还有更大的灵活性。当然，如果搜索的文本不仅是 Python，那么要搜索的其他文本也需要放入队列，一步步地传递进 `save_search_result` 中。

可以发现，爬完第 6 页协程就全部结束了。程序没有继续翻页直到爬完全部搜索页面才结束。怎么办呢？思路很简单，在页面抓取结束后发现有“下一页”，就把当前页之后的页数作为任务放进队列。但是需要注意，初始时放进了第 1 到第 6 页，假设第 1 页抓去完成后，应该放入第 2 页，但事实上第 2 页已经完成或者正在被其他协程抓取中。而初始时只放入第 1 页会造成协程池没有被充分利用。最简单的解决方案是保存当前抓取的页面和已经完成的页数的列表，保证不会重复抓取。可以想象，修改这个页数的列表是需要加锁的。

信号量（Semaphore）可以用来保证多协程（或者线程）并发访问共享资源时不会发生冲突。在访问开始时，协程必须获取一个信号量；访问结束后，该协程必须释放信号量，如果一个信号量的范围已经降低到 0，其他想访问的协程会被阻塞在 acquire 操作上，直到另一个已经获得信号量的协程做出释放。如果信号量设为 1，其实就是锁（Lock），表示是互斥访问，保证资源只在程序上下文被单次使用。我们使用锁来确认某页是否可以放入队列：

```
from gevent.coros import BoundedSemaphore
```

```
sem = BoundedSemaphore(1)
```

```
IN_POOL_TASKS = []
```

```
def put_new_page(p, queue):
    global IN_POOL_TASKS
    sem.acquire()
    sleep(0)
    if p not in IN_POOL_TASKS:
        queue.put(p)
        IN_POOL_TASKS.append(p)
    sem.release()
```

然后把程序中的 `queue.put(page)` 全部改成 `put_new_page(p, queue)`。

在函数 `save_search_result` 中添加爬取的其他页面的逻辑：

```
page_container = soup.find(id='pagebar_container')
if page_container and u'下一页' in page_container.text:
    last_page = int(page_container.find_all('a')[-2].text)
    current_page = int(page_container.find('span').text)
    for page in range(current_page + 1, last_page + 1):
        put_new_page(page, queue)
```

`put_new_page` 会把新的页面也放入队列，这样就可以可持续地翻页，直到没有下一页。

以一个优秀的爬虫标准来评价上述例子，还有如下两点需要改进：

- 微信搜索需要登录才能查询超过 10 页的结果，所以还需要实现登录功能。应该把登录和抓取步骤放在一个 `requests.Session` 中，保证整个过程在一个会话中完成。

- 没有支持增量爬取，当翻页时发现抓取到的文章之前已经抓取过，其实就不需要再往前翻页了。

使用多进程

使用进程代替线程可以有效避开 GIL，因为每个进程都拥有自己的 Python 解释器实例，也就不受 GIL 的限制了。计算密集型的任务通常应该使用多进程方式，也就是使用 multiprocessing 模块。multiprocessing 模块允许程序充分利用多处理器，并可以跨平台使用。I/O 密集型的任务瓶颈主要在网络延迟，所以使用多线程或者多进程都可以。

我们来实现解析微信文章页面的功能。首先从 Article 集合中找一篇微信公众号文章：

```
> db.article.find({'publisher': ObjectId("57317619bc1f551139de5b16")}, {'title': 1, 'url': 1, '_id': 0}).pretty()
{
  "title" : "Python Web 开发的十个框架",
  "url" : "http://mp.weixin.qq.com/s?src=3&timestamp=1462859281&ver=1&signature=2qvUQsQ6Tzf13kTij4VZ4cULEA7t1XgK8B6Ny*FKurB*HMwoUzEFYP CQyBE0g0XbGraIQxSJAoc-F8SGifgLH8t2JnKMF6B9tNcV87C6llfPcyfR98qbtQ 3J3KegBlGNo0sQYcGVKYRe9CMj6Pw5B0kjjlGC-tExRvECLLDThQs="
}
```

url 的值就是微信页面地址，微信公众号文章的页面上除了正文和配图，还展示了评论列表、阅读数和点赞数，这些内容不是 HTML 直接渲染的，而是通过 API 接口返回的，在 Chrome Developer Tools 中可以看到一个来至 <http://mp.weixin.qq.com/mp/getcomment> 的 API 请求，其中 src、timestamp、signature 和 ver 的参数值和 url 的是一样的，其他额外增加的参数除了空就是 0，只需要根据 url 构造一下就好了：

```
import urlparse
import urllib

COMMENT_JS_URL = 'http://mp.weixin.qq.com/mp/getcomment'

def gen_js_url(url):
    query_dct = urlparse.parse_qs(urlparse.urlsplit(url).query)
    query_dct = {k: v[0] for k, v in query_dct.items()}
    query_dct.update({'uin': '', 'key': '', 'pass_ticket': '', 'wxtoken': '',
                     'devicetype': '', 'clientversion': 0, 'x5': 0})
    return '{}?{}'.format(COMMENT_JS_URL, urllib.urlencode(query_dct))
```

爬取页面的意义就是抓取核心内容，然后在其他地方以某种方式展示出来，文章一般都会有配图，我们这里使用到“占位”这种技巧，在将来展示的时候再替换掉。举个例子，原来


```

    if comments:
        return comments[0]
    comment = cls(article=article, comment_id=comment_id, **kwargs)
    comment.save()
    return comment

```

在 Article 中的 comments 字段使用了 ListField(ReferenceField(Comment)), Comment 中的 article 字段使用的却是 ReferenceField('Article', dbref=True), 这是因为 Comment 是先定义的, 那个时候程序还没有从上到下地获取到 Article 这个模型, 所以需要使用字符串的 Article, 然后以设置 dbref 为 True 的方式实现互相引用。

给 Comment 添加的 get_or_create 方法是为了传入字段返回 comment 对象, 如没有对象则先创建再返回。

这次用到的 fetch 函数需要改了, 因为之前都是抓取单个页面, 现在是用一个会话连续抓取两个页面:

```
from simplejson.scanner import JSONDecodeError
```

```

def fetch(url):
    s = requests.Session()
    s.headers.update({'user-agent': get_user_agent()})
    proxies = {
        'http': Proxy.get_random()['address'],
    }
    html_text = s.get(url, timeout=TIMEOUT, proxies=proxies).text
    js_url = gen_js_url(url)
    try:
        js_data = s.get(js_url, timeout=TIMEOUT, proxies=proxies).json()
    except JSONDecodeError:
        raise RequestException()
    return html_text, js_data

```

在获取 API 接口的时候特意捕获 JSONDecodeError 的错误, 是因为 API 页面也有防爬策略。修改异常的错误类型是为了保证在外面调用 fetch 的时候只需要捕获 RequestException 一种异常就好了。

requests 使用了 simplejson, simplejson 其实就是标准库 json (Python 2.6 时加入) 的外部开发版本, 它更新很频繁且兼容 Python 2.5。使用上面的 API 返回内容测试一下:

```

In : %timeit -n 1000 json.dumps(api_result)
1000 loops, best of 3: 29.7 µs per loop
In : %timeit -n 1000 simplejson.dumps(api_result)
1000 loops, best of 3: 44.6 µs per loop

```

```
In : dumped = simplejson.dumps(api_result)
```

```
In : %timeit -n 1000 simplejson.loads(dumped)
```

```
1000 loops, best of 3: 44.1 µs per loop
```

```
In : %timeit -n 1000 json.loads(dumped)
```

```
1000 loops, best of 3: 86.1 µs per loop
```

可以看到 json 的 dump 操作更快, simplejson 的 loads 操作更快。

引用 json 最推荐采用如下的方式:

```
try:
```

```
    import simplejson as json
```

```
except ImportError:
```

```
    import json
```

获得文章评论列表的函数如下:

```
def get_comments(js_data, article):
```

```
    comments = []
```

```
    for comment in js_data['comment']:
```

```
        comment_id = comment['id']
```

```
        content = comment['content']
```

```
        create_at = datetime.fromtimestamp(float(comment['create_time']))
```

```
        nick_name = comment['nick_name']
```

```
        like_num = comment['like_num']
```

```
        comment = Comment.get_or_create(
```

```
            article, comment_id, content=content, create_at=create_at,
```

```
            nick_name=nick_name, like_num=like_num)
```

```
        comments.append(comment)
```

```
    return comments
```

更新文章的函数如下:

```
def update_article(article, html_text, js_data):
```

```
    soup = BeautifulSoup(html_text, 'lxml')
```

```
    p_contents = soup.find(class_='rich_media_content').find_all('p')
```

```
    content = []
```

```
    pictures = {}
```

```
    picture_count = 1
```

```
    for p_content in p_contents:
```

```
        img = p_content.find('img')
```

```
        if img is None:
```

```
            content.append(p_content.text.encode('utf-8'))
```

```
        else:
```

```
            tag = '<图片{}>'.format(picture_count)
```

```
            content.append(tag)
```

```

        pictures[tag] = img['data-src']
        picture_count += 1

    article.content = '\n'.join(content)
    article.pictures = pictures
    article.comments = get_comments(js_data, article)
    article.like_num = js_data['like_num']
    article.read_num = js_data['read_num']
    article.save()
    return article

```

按照 13.2 节的思路，我们先创建保存文章结果的函数：

```

def save_article_result(article, queue=None, retry=0):
    url = article.article_url

    try:
        html_text, js_data = fetch(url)
    except RequestException:
        retry += 1
        if retry > 5:
            queue.put(url)
            return
        return save_article_result(article, queue, retry)
    return update_article(article, html_text, js_data)

```

其中参数 `article` 就是一个 `Article` 对象。

现在前期准备都完成了，接下来使用多进程和队列来更新 `article` 集合中存在的文章。多进程方式不再使用 `Queue` 模块，而是换成 `multiprocessing.Queue` 或者 `multiprocessing.JoinableQueue`。`JoinableQueue` 有 `join` 和 `task_done` 两个方法，所以适合存放统一的待执行任务，供并发进程获取。而 `Queue` 可以用来收集任务执行的结果：

```

import multiprocessing

def use_multiprocessing_with_queue():
    queue = multiprocessing.JoinableQueue()
    num_consumers = multiprocessing.cpu_count() * 2

    for article in Article.objects.all():
        queue.put(article)

    for _ in range(num_consumers):
        p = multiprocessing.Process(target=save_article_result_with_queue,
                                    args=(queue,))

```

```
p.start()

queue.join()
```

通常计算密集型的程序使用的线程和进程数量一般与 CPU 核数一致，I/O 密集型的程序使用的线程和进程数量一般是 CPU 核数的 2 倍。也可以根据网络情况，使用 3 倍甚至更多的倍数，在实际工作中通过测试获取最佳值。

看一下消费者进程需要执行的函数：

```
from Queue import Empty

from mongoengine.connection import disconnect

from models import Article, lazy_connect

def save_article_result_with_queue(queue):
    disconnect()
    lazy_connect()
    while 1:
        try:
            article = queue.get(timeout=1)
        except Empty:
            break
        save_article_result(article, queue)
        queue.task_done()
```

多进程队列为空时抛的异常是 `Queue.Empty`。这里用“先关闭 MongoDB 连接，再使用 lazy 模式连接”的原因，是为了解决 MongoDB 使用多进程时，调用 `getaddrinfo` 可能会发生的死锁（<http://bit.ly/1sHScFY>）的问题。

`lazy_connect` 在 `models.py` 中的定义如下：

```
from config import DB_HOST, DB_PORT, DATABASE_NAME

def lazy_connect():
    connect(DATABASE_NAME, host=DB_HOST, port=DB_PORT)

lazy_connect()
```

在 13.1 节，演示了通过第二个队列存放第一个队列的执行结果，最后获得抓取总数的方法。本节我们再简化一些，不用创建第二个队列就能获得全部修改后的 `Article` 模型：

```
def save_article_result_with_queue2(in_queue, out_queue):
    while 1:
        try:
            article = in_queue.get(timeout=1)
        except Empty:
            break
        updated_article = save_article_result(article, in_queue)
        out_queue.put(updated_article) # 把更新好的文章对象放入out_queue这个队列中
    in_queue.task_done()
```

使用两个队列的多进程主程序如下：

```
def use_multiprocessing_with_queue2():
    queue = multiprocessing.JoinableQueue()
    num_consumers = multiprocessing.cpu_count() * 2
    results_queue = multiprocessing.Queue()

    for article in Article.objects.all()[5:8]:
        queue.put(article)

    for _ in range(num_consumers):
        p = multiprocessing.Process(target=save_article_result_with_queue2,
                                    args=(queue, results_queue))
        p.start()

    queue.join()

    results = []

    while 1:
        try:
            updated_article = results_queue.get(timeout=1)
        except Empty:
            break
        results.append(updated_article)
    print len(results)
```



放入队列的对象只要是可 pickle 化的就可以，所以 Article 对象可以作为消息放入队列。

如果文章页面是动态注入的，需要使用 PhantomJS 之类的工具模拟页面打开的过程，再从加载完毕的页面文件中进行解析。

使用 Future

`concurrent.futures` 是 Python 3.2 引入的用于处理并发的模块，它提供一种优雅的用多线程或者多进程实现并发的方式。Python 2 中也有移植的对应包，可以使用如下命令安装：

```
> pip install futures
```

`concurrent.futures` 中包含 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 这两个执行器，分别产生线程池和进程池。我们把清理不可用代理的代码改成使用 `ThreadPoolExecutor` 模式：

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
```

```
def check_proxy(p):
    try:
        fetch('http://baidu.com', proxy=p['address'])
    except RequestException:
        p.delete()
        return False
    return True
```

```
with ThreadPoolExecutor(max_workers=5) as executor:
    for p in Proxy.objects.all():
        executor.submit(check_proxy, p)
```

`max_workers` 参数定义了供执行器使用的线程数量。`executor.submit` 表示手动提交单个任务，`submit` 执行的返回值是一个 `concurrent.futures.Future` 实例，如果调用 `Future` 实例的 `result` 方法就可以获得结果，结果被返回之前进程是阻塞的。还可以给 `Future` 实例添加回调函数：

```
from functools import partial
```

```
def when_done(p, f):
    print '[{}]: {}'.format(p.address, 'succeed' if f.result() else 'failure')
```

```
def use_thread_pool_executor_with_cb():
    with ThreadPoolExecutor(max_workers=5) as executor:
        for p in Proxy.objects.all():
            result = executor.submit(check_proxy, p)
            result.add_done_callback(partial(when_done, p))
```

`Future` 类还提供取消任务、检查任务状态（运行中或者已完成）等功能，也可以汇报执行结果或者执行时发生的异常。

这样就能在检查完每个代理时打印出代理是否可用。由于 `add_done_callback` 方法只接受一个函数，这个函数只接受一个参数，也就是 `Future` 实例，而我们这个例子传入了两个参数，所以使用了 `partial` 将 `p` 预先传到函数中。

`ProcessPoolExecutor` 和 `ThreadPoolExecutor` 的接口一样，接下来修改成使用多进程执行器，并用 `map` 方法并行提交任务：

```
with ProcessPoolExecutor(max_workers=5) as executor:
    executor.map(check_proxy, Proxy.objects.all())
```

最后使用 `Queue` 和 `ProcessPoolExecutor` 改写 13.3 节抓取微信文章的功能。爬取的逻辑还是用之前实现的 `save_article_result`。由于在 `save_article_result` 中可能因抓取异常造成重试，超过阈值时会将文章重新放入队列，而在 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 中队列被封装了，之前是我们自己维护队列，现在需要借用封装好的队列，其中 `ThreadPoolExecutor` 的任务队列存放在 `executor._work_queue`：

```
from functools import partial

from concurrent.futures import ThreadPoolExecutor

from save_article_content import save_article_result
from models import Article

with ThreadPoolExecutor(max_workers=5) as executor:
    executor.map(partial(save_article_result, queue=executor._work_queue),
                 Article.objects.all())
```

上述的实现是不是比之前的版本要简洁得多呢？

如果希望获取执行的结果可以使用 `as_completed`：

```
from concurrent.futures import as_completed

with ProcessPoolExecutor(max_workers=5) as executor:
    future_tasks = {executor.submit(
        check_proxy, p): p for p in Proxy.objects.all()}
    for future in as_completed(future_tasks):
        p = future_tasks[future]
        try:
            rs = future.result()
        except Exception as exc:
            print 'Receive exception: {}'.format(exc)
        else:
            print '[{}]: {}'.format(
```

```
p.address, 'succeed' if rs else 'failure')
```

这样就可以把结果收集起来，不再需要使用两个队列了。

使用 asyncio

Python 2 对协程的支持是通过生成器（Generator）实现的。利用 yield 实现生产者/消费者模型的例子如下（use_yield.py）：

```
import random

def consumer():
    r = None
    while 1:
        data = yield r
        print 'Consuming: {}'.format(data)
        r = data + 1

def producer(consumer):
    n = 3
    consumer.send(None)
    while n:
        data = random.choice(range(10))
        print('Producing: {}'.format(data))
        rs = consumer.send(data)
        print 'Consumer return: {}'.format(rs)
        n -= 1
    consumer.close()

c = consumer()
producer(c)
```

执行的结果如下：

```
> python chapter13/section5/use_yield.py
Producing: 0
Consuming: 0
Consumer return: 1
Producing: 4
Consuming: 4
Consumer return: 5
Producing: 6
```

```
Consuming: 6
Consumer return: 7
```

`c.send(None)` 和 `c.next()` 的作用一样，都是让协程运行起来，通过之后不断 `send` 数据给消费者，消费者再通过 “`yield r`” 把执行结果返回给发布者这样的方式，实现一个生产/消费循环。

Python 2 内置的支持大抵如此。大量的项目（尤其是网络编程相关的项目）都是通过使用第三方库实现的协程来编写程序，如 `Eventlet` 和 `Gevent`。由于 Python 2 语言的局限，协程的实现比较原始，众多第三方库的实现并不统一，并且通常都需要使用类似 `Monkey Patch` 的技术才能实现非阻塞 I/O 等特性来真正提高性能。

使用 `yield` 语句只能将 CPU 控制权还给直接调用者，Python 3.3 中添加了 “`yield from`” 表达式，允许生成器它的部分操作委任给另外一个生成器。但是仍有一些缺点：

- 协程与常规的生成器使用相同语法时容易混淆，尤其对于新的开发者而言。
- 一个函数是否是协程需要通过主体代码中是否使用了 `yield` 或者 `yield from` 语句进行检测，这一点容易在重构中忽略，而导致迷惑和错误。
- 对异步调用的支持被 `yield` 的语法限制了，不能使用更多的语法特性，比如 `with` 和 `for`。

Python 3.4 中 `asyncio` 被纳入了标准库，它提供了使用协程编写单线程并发代码，通过 I/O 多路复用访问套接字和其他资源，运行网络客户端和服务器等原语。而 Python 3.5 添加了 `async` 和 `await` 这两个关键字。自此，协程成为新的语法，而不再是一种生成器类型了。I/O 多路复用与协程的引入，可以极大提高高负载下程序的 I/O 性能。

async/await

`async` 用于声明一个协程：

```
async def foo():
    pass
```

在普通的函数前加上 `async` 关键字后，这个函数就变成了一个协程。

`await` 表示等待另一个协程执行完返回，获取协程执行结果，它必须在协程内才能使用。

Python 3.5 之前协程是这样写的（`old_coroutine.py`）：

```
import asyncio
```

```
@asyncio.coroutine
def slow_operation(n):
```

```
    yield from asyncio.sleep(1)
    print('Slow operation {} complete'.format(n))

@asyncio.coroutine
def main():
    yield from asyncio.wait([
        slow_operation(1),
        slow_operation(2),
        slow_operation(3),
    ])

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

asyncio 事件循环受到了 Tornado 与 Twisted 等的影响，使用事件循环，Tornado、Twisted 以及 Gevent 可以与 asyncio 一起工作。asyncio 还为每个平台选择了最佳的 I/O 机制，比如 UNIX 和 Linux 平台上使用 selectors 库来做系统级别的 I/O 切换。

调用 `get_event_loop` 将返回默认的事件循环，用于负责所有协程的调度。在大量协程并发执行的过程中，除了在协程中主动使用 `await`，当本地协程发生 I/O 等待时，调用 `asyncio.sleep`，程序的控制权也会在不同的协程间切换，从而在 GIL 的限制下实现最大程度的并发执行，不会由于等待 I/O 等原因导致程序阻塞，达到较高的性能。

执行结果如下：

```
> time python3 chapter13/section5/old_coroutine.py
Slow operation 2 complete
Slow operation 3 complete
Slow operation 1 complete
python3 chapter13/section5/old_coroutine.py  0.10s user 0.05s system 12% cpu 1.201
total
```



接下来的例子都使用 Python 3.5.1 执行。

使用 `async/await` 关键词之后会是这样（`new_coroutine.py`）：

```
import asyncio

async def slow_operation(n):
    await asyncio.sleep(1)
    print('Slow operation {} complete'.format(n))
```

```
async def main():
    await asyncio.wait([
        slow_operation(1),
        slow_operation(2),
        slow_operation(3),
    ])

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

可以感受到，`async` 的使用简化了 `asyncio.coroutine`；`await` 的使用简化了 `yield from`。

除了“`async def`”，还有“`async for`”和“`async with`”关键字。

1. `async for`：异步迭代器语法。为了支持异步迭代，异步对象需要实现 `__aiter__` 方法，异步迭代器需要实现 `__anext__` 方法，停止迭代需要在 `__anext__` 方法内抛出 `StopAsyncIteration` 异常（`async_for.py`）。

```
import random
import asyncio

class AsyncIterable:
    def __init__(self):
        self.count = 0

    async def __aiter__(self):
        return self

    async def __anext__(self):
        if self.count >= 5:
            raise StopAsyncIteration
        data = await self.fetch_data()
        self.count += 1
        return data

    async def fetch_data(self):
        return random.choice(range(10))

async def main():
    async for data in AsyncIterable():
        print(data)
```

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

2. `async with`: 异步上下文管理器语法。为了支持上下文管理器，需要实现 `__aenter__` 和 `__aexit__` 方法（`async_with.py`）。

```
async def log(msg):
    print(msg)

class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')

async def coro():
    async with AsyncContextManager():
        print('body')
```

除了使用事件循环，采用原来的 `send(None)` 方式也是可以的：

```
c = coro()

try:
    c.send(None)
except StopIteration:
    print('finished')
```

它的执行结果如下：

```
entering context
body
exiting context
finished
```

Future

`Future` 是一种异步编程范式，它对异步过程调用的结果做了抽象，它并不关心具体的异步机制。无论是线程、网络，还是 I/O，甚至 RPC，只要是异步过程调用，都可以通过 `Future` 的概念统一处理。基于 `Future` 的接口可以简化代码编写，让各种异步操作以一种顺序的、更接近人类逻辑思维的方式编写异步代码。

asyncio.Future 几乎兼容 13.4 节介绍的 concurrent.futures.Future:

```
> cat chapter13/section5/async_future.py
async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
print('Future Done: {}'.format(future.done()))
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print('Future Done: {}'.format(future.done()))
print(future.result())
loop.close()
```

它的执行结果如下:

```
Future Done: False
Future Done: True
Done!
```

asyncio.Future 也支持添加回调函数:

```
from functools import partial

def set_result(future, result):
    print('Setting future result to {!r}'.format(result))
    future.set_result(result)

def callback(who, future):
    print('[{}]: returned result: {!r}'.format(who, future.result()))

event_loop = asyncio.get_event_loop()
future = asyncio.Future()
future.add_done_callback(partial(callback, 'CB1'))
event_loop.call_soon(set_result, future, 'Done!')
future.add_done_callback(partial(callback, 'CB2'))
event_loop.run_until_complete(future)
event_loop.close()
```

通过 call_soon 添加修改结果的回调函数, 通过 add_done_callback 添加 Future 完成的回调函数, 它的执行结果如下:

```
Setting future result to 'Done!'
[CB1]: returned result: 'Done!'
[CB2]: returned result: 'Done!'
```

可见添加的回调是一个先进先出（FIFO）的队列，保证了回调的顺序。这里强调一点，Future 的结果在设置之后就不能修改了：

```
...
future.add_done_callback(partial(callback, 'CB1'))
event_loop.call_soon(set_result, future, 'Done!')
event_loop.call_soon(set_result, future, 'Done again!')
event_loop.run_until_complete(future)
```

如果再将结果设置为 “Done again!”，就会收到 `InvalidStateError` 错误。

使用 aiohttp

现在把 13.4 节的抓取微信文章页面的爬虫用 `asyncio` 来实现。首先创建一个 Python 3 的虚拟环境，再安装相关依赖：

```
> pyenv-3.5 ~/venv3
> source ~/venv3/bin/activate
> pip install beautifulsoup4 lxml aiohttp mongengine fake_useragent cchardet
```

`chardet` 是一个常用的字符编码检测器，`cchardet` 是一个更快的实现，可以替代 `chardet`。

用 `aiohttp` 替代 `requests` 作为 HTTP 客户端，先感受下 `aiohttp` 的用法：

```
from asyncio import TimeoutError

import aiohttp
from aiohttp import ProxyConnectionError

async def fetch(retry=0):
    proxy = 'http://{}/{}'.format(Proxy.get_random()['address'])
    headers = {'user-agent': get_user_agent()}
    conn = aiohttp.ProxyConnector(proxy=proxy)

    url = 'http://httpbin.org/ip'

    try:
        with aiohttp.ClientSession(connector=conn) as session:
            with aiohttp.Timeout(TIMEOUT):
                async with session.get(url, headers=headers) as resp:
                    return await resp.json()
```



```

except (ProxyConnectionError, TimeoutError):
    try:
        p = Proxy.objects.get(address=proxy)
        if p:
            p.delete()
    except DoesNotExist:
        pass
    retry += 1
    if retry > 5:
        raise TimeoutError()
    await asyncio.sleep(1)
    return await fetch(retry=retry)

```

requests 所支持的常用特性 aiohttp 也都是支持的，只是在用法上有比较大的调整。

使用事件循环调用 fetch 函数：

```

loop = asyncio.get_event_loop()
f = asyncio.wait([fetch()])
completed, pending = loop.run_until_complete(f)

for future in completed:
    print(future.result())

```

其中使用 asyncio.wait 接受了一个任务列表，run_until_complete 返回的也会是一个任务结果的列表。

aiohttp 也可以用来实现 HTTP 服务，我们实现一个简单的 API 服务（aiohttp_server.py）：

```

import json
import asyncio

import aiohttp
from aiohttp import web

REQUEST_URLS = [
    'http://httpbin.org/ip',
    'http://httpbin.org/user-agent',
    'http://httpbin.org/headers'
]

async def handle(request):
    coroutines = [aiohttp.request('get', url) for url in REQUEST_URLS]

    results = await asyncio.gather(*coroutines, return_exceptions=True)

```

```
response_data = {
    url: not isinstance(result, Exception) and result.status == 200
    for url, result in zip(REQUEST_URLS, results)
}

body = json.dumps(response_data).encode('utf-8')
return web.Response(body=body, content_type="application/json")

loop = asyncio.get_event_loop()
app = web.Application(loop=loop)
app.router.add_route('GET', '/', handle)

server = loop.create_server(app.make_handler(), '0.0.0.0', 8080)
print('Server started at http://127.0.0.1:8080')
loop.run_until_complete(server)
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
```

使用 `asyncio.gather` 替代 `asyncio.wait` 并设置 `return_exceptions` 为 `True`，会收集到全部内容，而且不会抛出异常，只是返回一个异常的对象。

使用队列

`asyncio` 同样支持多种原语，如锁（Lock）、事件（Event）、信号量（Semaphore）、条件变量（Condition），当然也支持队列。`asyncio` 中的队列的设计很像 `Queue` 模块，但是没有 `timeout` 参数，只能通过 `asyncio.wait_for` 在任务超时之后取消任务。队列包含如下三种。

- `Queue`：用生产者/消费者模型的队列，适合我们的微信文章抓取队列。
- `PriorityQueue`：`Queue` 的子类，带有优先级的队列。
- `LifoQueue`：`Queue` 的子类，最近添加优先的队列。

改写微信抓取，其实就是做如下四件事。

1. 修改使用的标准库模块的引用方法。比如 `url` 解析相关的函数都放在 `urllib.parse` 下，需要修改为如下引用：

```
from urllib.parse import urlparse, urlsplit, parse_qs, urlencode
```
2. 给希望改写成协程的函数添加 `async` 关键字。

3. 在调用协程函数的地方添加 `await` 关键字。
4. 使用事件循环。

看一下修改后的 `save_article_result_with_queue` 函数：

```
async def save_article_result_with_queue(queue):
    while 1:
        article = await queue.get()
        if article is None:
            queue.task_done()
            break
        await save_article_result(article, queue)
        queue.task_done()
```

和之前的处理空队列的逻辑不同，我们没有捕获 `Empty` 异常然后 `break`，而是向队列添加一个 `None`，`get` 的时候发现消息为 `None` 就结束循环。这样做的原因是 `queue.get` 不接受 `timeout` 参数，这样可以简化实现。

把文章放入队列的操作，我们放在一个协程函数中来做：

```
async def producer(queue):
    for article in Article.objects.all():
        await queue.put(article)

    for i in range(5): # 有几个协程就放为几个为None的消息
        await queue.put(None)

    await queue.join()
```

现在使用事件循环：

```
loop = asyncio.get_event_loop()
queue = asyncio.Queue()
consumers = [
    loop.create_task(save_article_result_with_queue(queue,))
    for i in range(5)
]
prod = loop.create_task(producer(queue))
loop.run_until_complete(
    asyncio.wait(consumers + [prod])
)
```

`create_task` 创建的是 `asyncio.tasks.Task` 对象，它是 `asyncio.Future` 的子类，相当于 `asyncio` 对 `Future` 的封装。

第 14 章

Python 进阶

本章主要包含如下内容：

- 介绍 `errno`、`subprocess`、`contextlib`、`glob`、`operator`、`functools`、`collections` 模块的使用方法。
- 笔者对《Python 之禅》的理解。
- 笔者总结的一些 Python 实践经验，并列举了两篇最佳实践的文章。
- 介绍一些 Python 3 的有用功能，并移植到 Python 2。
- 通过真实的例子演示如何使用 CFFI/Cython 编写 Python 扩展，并对比二者的执行效率。
- 演示使用 PyObjC 发送通知的例子，让开发者收到的通知更有针对性。

使用标准库模块

笔者在面试产品开发人员的时候，都会和面试者聊一聊其工作中常用的几个 Python 标准库模块的作用和功能（像 `os`、`sys` 这样的模块除外），这其实从侧面反映了一个工程师对技术的态度和对 Python 的熟悉程度。

Python 内置了非常多的模块，大部分是纯 Python 的，也有一些用 C 编写的。它们提供标准化的解决方案去解决日常编程中出现的许多问题，建议熟读这些标准库代码。很多最佳实践的代码实现都在标准库里面可以找到，不需要自己造轮子，你也可以参考标准库的实现再去扩展功能。举个例子，笔者之前经常需要将一个多层嵌套的数组（嵌套可以是任何层数）转换为只有一层的数组，也就是编写一个函数 `flatten`，让 `flatten([1, [2], [3, [[4]]])` 的执

行结果是 [1, 2, 3, 4]。在不同的项目中都要把笔者实现的这个 `flatten` 函数放进去。后来发现其实标准库中已经有非常好的实现了：

```
In : from compiler.ast import flatten
In : flatten([1, [2], [3, [[4]]]])
Out: [1, 2, 3, 4]
```

本节将介绍一些非常有用的标准库模块。

errno

在日常开发中经常需要捕获各种异常，做特殊处理。举个例子：

```
In : os.kill(12345, 0)
-----
OSError                                Traceback (most recent call last)
<ipython-input-32-11dc6caa503c> in <module>()
----> 1 os.kill(12345, 0)
```

```
OSError: [Errno 3] No such process
```

信号为 0，表示这只是检查 PID 的有效性。根据提示，这是一个 “No such process” 类型的错误，注意方括号中的 “Errno 3”，这其实是 Python 内置的错误系统提供的编号：

```
In : os.strerror(3)
Out: 'No such process'
```

还可以使用 `errno` 模块找到对应的错误类型，更精准地做异常处理（`errno_example.py`）：

```
import os
import errno

def listdir(dirname):
    try:
        os.listdir(dirname)
    except OSError as e:
        error = e.errno
        if error == errno.ENOENT:
            print 'No such file or directory'
        elif error == errno.EACCES:
            print 'Prmission denied'
        elif error == errno.ENOSPC:
            print 'No space left on device'
        else:
            print e.strerror
```

```
    else:
        print 'No error!'

for filename in ['/no/such/dir', '/root', '/home/ubuntu']:
    listdir(filename)
```

通过对比异常对象的 `errno` 属性值就能知道异常类型。

subprocess

`subprocess` 模块用来取代如下模块和函数：

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

`subprocess` 模块提供了几个有用的方法。

1. `call`：执行系统命令，可以替代 `os.system`。`call` 只返回命令的返回值。

```
In : subprocess.call('ls -l /tmp/mongodb-27017.sock', shell=True)
srwx----- 1 mongodb mongodb 0 May 10 12:24 /tmp/mongodb-27017.sock
Out: 0
In : subprocess.call('exit 1', shell=True)
Out: 1
```

通常由于安全问题，不建议使用 `shell = True`，可以把命令拆分成列表：

```
In : subprocess.call(['ls', '/tmp/mongodb-27017.sock'], shell=False)
/tmp/mongodb-27017.sock
Out: 0
```

拆分命令最简单的方法是使用 `shlex` 模块：

```
In : shlex.split('ls /tmp/mongodb-27017.sock')
Out: ['ls', '/tmp/mongodb-27017.sock']
```

2. `check_call`：添加了错误处理的执行系统命令方法。当执行 `call` 方法的返回值不为 0，就会抛出 `CalledProcessError` 异常。

```
In : subprocess.check_call("exit 1", shell=True)
-----
CalledProcessError                                Traceback (most recent call last)
<ipython-input-14-1f03a695d5c6> in <module>()
```

```
----> 1 subprocess.check_call("exit 1", shell=True)
...
CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

3. Popen: 一个用来执行子进程的类, 通过 `communicate` 方法获得执行结果:

```
In : from subprocess import Popen, PIPE
In : proc = Popen(['echo', 'hello!'], stdout=PIPE)
In : stdout, stderr = proc.communicate()
In : print stdout
hello!
```

Popen 类经常用来实现 Shell 的管道功能, 假设要执行 “`ls /tmp | grep mongodb`”, 可以使用如下方式:

```
In : p1 = Popen(['ls', '/tmp'], stdout=PIPE)
In : p2 = Popen(['grep', 'mongodb'], stdin=p1.stdout, stdout=PIPE)
In : stdout, _ = p2.communicate()
In : stdout
Out: 'mongodb-27017.sock\n'
```

4. `check_output`: 在 Python 2.7 和 Python 3 中都可用, 它比 Popen 更简单地获得输出, 但是需要执行的返回值为 0, 否则仍然抛出 `CalledProcessError` 异常。

```
In : subprocess.check_output(['ls', '-l', '/tmp/mongodb-27017.sock'])
Out: 'srwx----- 1 mongodb mongodb 0 May 10 12:24 /tmp/mongodb-27017.sock\n'
```

在出现错误的时候, 可以额外地执行 `exit 0`, 就能正常获得输出:

```
In : subprocess.check_output('ls /no_such_file; exit 0', stderr=subprocess.
    STDOUT, shell=True)
Out: 'ls: cannot access /no_such_file: No such file or directory\n'
```

contextlib

写 Python 代码的时候经常将一系列操作放在一个语句块中, Python 2.5 加入了 `with` 语法, 实现上下文管理功能, 这让代码的可读性更强并且错误更少。最常见的例子就是 `open`, 如果不使用 `with`, 使用 `open` 时会是这样:

```
In : f = open('/tmp/a', 'a')
In : f.write('hello world')
In : f.close()
```

如果使用 `with`, 可以简化为两行:

```
In : with open('/tmp/a', 'a') as f:
```

```
....:     f.write('hello world')
....:
```

在执行完缩进的代码块后会自动关闭文件。

同样的例子还有 `threading.Lock`，如果不使用 `with`，需要这样写：

```
import threading
lock = threading.Lock()

lock.acquire()
try:
    my_list.append(item)
finally:
    lock.release()
```

如果使用 `with`，就会非常简单：

```
with lock:
    my_list.append(item)
```

创建上下文管理器实际就是创建一个类，添加 `__enter__` 和 `__exit__` 方法。看看 `threading.Lock` 的上下文管理功能是怎么实现的：

```
class LockContext(object):

    def __init__(self):
        print '__init__'
        self.lock = threading.Lock()

    def __enter__(self):
        print '__enter__'
        self.lock.acquire()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print '__exit_'
        self.lock.release()

with LockContext():
    print 'In the context'
```

执行的输出如下：

```
__init__
__enter__
In the context
__exit_
```


上面的例子比较简单，在执行 `self.__enter__` 的时候没有传递参数。下面我们来实现 `open` 的上下文管理功能：

```
class OpenContext(object):

    def __init__(self, filename, mode):
        self.fp = open(filename, mode)

    def __enter__(self):
        return self.fp

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.fp.close()

with OpenContext('/tmp/a', 'a') as f:
    f.write('hello world')
```

自定义上下文管理器确实很方便，但是 Python 标准库还提供了更易用的上下文管理器工具模块 `contextlib`，它是通过生成器实现的，我们不必再创建类以及 `__enter__` 和 `__exit__` 这两个特殊的方法：

```
from contextlib import contextmanager

@contextmanager
def make_open_context(filename, mode):
    fp = open(filename, mode)
    try:
        yield fp
    finally:
        fp.close()

with make_open_context('/tmp/a', 'a') as f:
    f.write('hello world')
```

`yield` 关键词把上下文分割成两部分：`yield` 之前就是 `__init__` 中的代码块；`yield` 之后其实就是 `__exit__` 中的代码块；`yield` 生成的值会绑定到 `with` 语句 `as` 子句中的变量（如果没有生成，也就没有 `as` 字句）。

如果有多个上下文，通常需要嵌套：

```
@contextmanager
def make_context(*args):
    print args
```

```
yield
```

```
with make_context(1, 2) as A:
    with make_context(3, 4) as B:
        print 'In the context'
```

事实上，Python 2.7 中 with 语句不需要嵌套：

```
with make_context(1, 2) as A, make_context(3, 4) as B:
    print 'In the context'
```

如果用的 Python 版本小于 2.7，需要使用 contextlib.nested 减少 with 语句的嵌套：

```
from contextlib import nested
```

```
with nested(make_context(1, 2), make_context(3, 4)) as (A, B):
    print 'In the context'
```

glob

glob 用来匹配 UNIX 风格的路径名字模块，它支持 “*”、“?”、“[]” 这三种通配符，“*” 代表 0 个或多个字符，“?” 代表一个字符，“[]” 匹配指定范围内的字符，例如 [0-9] 匹配数字。我们先创建一个目录，目录中包含如下文件：

```
> ls /tmp/chapter14
1.txt 2.txt a2.txt a3.txt a.txt b.txt
```

看一下 glob 的匹配效果：

```
In : glob.glob('/tmp/chapter14/a*.txt')
Out: ['/tmp/chapter14/a3.txt', '/tmp/chapter14/a.txt', '/tmp/chapter14/a2.txt']
In : glob.glob('/tmp/chapter14/a?.txt')
Out: ['/tmp/chapter14/a3.txt', '/tmp/chapter14/a2.txt']
In : glob.glob('/tmp/chapter14/a[0-9].txt')
Out: ['/tmp/chapter14/a3.txt', '/tmp/chapter14/a2.txt']
In : glob.glob('/tmp/chapter14/a[0-2].txt')
Out: ['/tmp/chapter14/a2.txt']
```

operator

operator 是一个内建操作的函数式接口。举个例子，如果想对列表的值求和，可以使用 sum，但是如果要求相乘呢？其实可以结合 reduce 和 operator 模块来实现：

```
In : import operator
In : reduce(operator.mul, (5, 4, 3, 2, 1))
Out: 120
```

还能借用 operator 模块实现伪 lisp 代码:

```
In : def f(op, *args):
...:     return {
...:         '+' : operator.add,
...:         '-' : operator.sub,
...:         '*' : operator.mul,
...:         '/' : operator.div,
...:         '%' : operator.mod
...:     }[op](*args)
...:
In : f('*', f('+', 1, 2), 3)
Out: 9
```

operator 模块还提供了非常有用的 itemgetter、attrgetter 和 methodcaller 方法。

1. itemgetter。通过被求值对象的 __getitem__ 方法获得符合的条目:

```
In : l = [1, 2, 3, 4, 5]
In : operator.itemgetter(1)(l)
Out: 2
In : operator.itemgetter(1, 3, 4)(l)
Out: (2, 4, 5)
```

另一个使用 itemgetter 的场景是排序:

```
In : objs = [('a', 2), ('b', 4), ('c', 1)]

In : sorted(objs, key=operator.itemgetter(1))
Out: [('c', 1), ('a', 2), ('b', 4)]
```

上述例子是对列表每个元组的第二个元素进行升序排序, 等价于:

```
In : sorted(objs, key=lambda x: x[1])
Out: [('c', 1), ('a', 2), ('b', 4)]
```

2. attrgetter。attrgetter 根据被求值对象的属性获得符合条件的结果:

```
In : import sys
In : operator.attrgetter('platform')(sys)
Out: 'linux2'
In : operator.attrgetter('platform', 'maxint')(sys)
Out: ('linux2', 9223372036854775807)
```

想要获得 a.b.c.d 这样的层级很深的属性或者想动态获得嵌套属性，若不使用 attrgetter，则写法是：

```
In : attr = a
```

```
In : for sub in ['b', 'c', 'd']:
....:     attr = getattr(attr, sub)
....:
```

再简单一点：

```
In : reduce(getattr, 'b.c.d'.split('.'), a)
```

而使用 attrgetter 就很简洁：

```
In : operator.attrgetter('b.c.d')(a)
```

3. methodcaller。methodcaller 将调用被求值对象的方法：

```
# 相当于h.hello()
operator.methodcaller('hello')(h)
# 相当于h.func(1, b=2)
operator.methodcaller('func', 1, b=2)(h)
```

functools

functools 模块中包含了一系列操作其他函数的工具。

1. partial。partial 可以重新定义函数签名，也就是在执行函数之前把一些参数预先传给函数，待执行时传入的参数数量会减少。

```
In : import functools
```

```
In : def f(a, b=3):
....:     return a + b
....:
In : f2 = functools.partial(f, 1)
In : f2(5)
Out: 6
In : f2()
Out: 4
```

上例中，本来函数 f 需要传入两个参数，但是使用 partial 把参数 a 传入（值为 1），生成了函数 f2，调用 f2 只需要传入参数 b（或者不传入，直接使用默认值）即可。当然可以在 partial 的时候把参数 b 的值预先传入：

```
In : f2 = functools.partial(f, b=10)
In : f2(1)
Out: 11
```

2. `wraps`。把被封装函数的 `__name__`、`__module__`、`__doc__` 和 `__dict__` 复制到封装函数中，这样在未来排错或者函数自省的时候能够获得正确的源函数的对应属性，所以使用 `wraps` 是一个好习惯。我们先看不使用 `wraps` 的例子：

```
In : def deco(f):
...:     def wrapper(*args, **kwargs):
...:         return f(*args, **kwargs)
...:     return wrapper
...:
In : @deco
...: def func():
...:     '''This is __doc__'''
...:     return 1
...:
In : func.__doc__
In : func.__name__
Out: 'wrapper'
```

可以发现 `func.__doc__` 和 `func.__name__` 等都是错误的，它们错误地使用装饰器的对应属性。正确的方法是使用 `wraps` 拷贝源函数属性：

```
In : def deco2(f):
...:     @functools.wraps(f)
...:     def wrapper(*args, **kwargs):
...:         return f(*args, **kwargs)
...:     return wrapper
...:
In : @deco2
...: def func():
...:     '''This is __doc__'''
...:     return 1
...:
In : func.__doc__
Out: 'This is __doc__'
In : func.__name__
Out: 'func'
```

3. `total_ordering`。对比自定义对象需要添加 `__lt__`、`__le__`、`__gt__`、`__ge__` 和 `__eq__` 等方法，如果使用 `total_ordering`，只需要定义 `__eq__`，以及定义 `__lt__`、`__le__`、`__gt__`、`__ge__` 四种方法之一就可以了：

```
In : @functools.total_ordering
...: class Size(object):
...:     def __init__(self, value):
...:         self.value = value
...:     def __lt__(self, other):
...:         return self.value < other.value
...:     def __eq__(self, other):
...:         return self.value == other.value
```

```
In : Size(3) > Size(2)
Out: True
In : Size(2) == Size(2)
Out: True
```

`total_ordering` 会自动设置未定义的三种特殊方法。

4. `cmp_to_key`。Python 2 的 `sorted` 函数除了通过指定 `key` 参数的值作为依据来排序，还支持 `cmp` 参数：

```
In : def numeric_compare(x, y):
...:     return x[1] - y[1]
...:
In : sorted(objs, cmp=numeric_compare)
Out: [('c', 1), ('a', 2), ('b', 4)]
```

`cmp_to_key` 可以帮助我们将这种旧式比较的 `cmp` 函数转换为 `key` 的参数：

```
In : sorted(objs, key=functools.cmp_to_key(numeric_compare))
Out: [('c', 1), ('a', 2), ('b', 4)]
```

除了兼容 Python 3 的 `sorted`，还可以把比较函数转换后用于 `min`、`max`、`heapq.nlargest`、`heapq.nsmallest`、`itertools.groupby` 等支持 `key` 参数的函数上。

collections

`collections` 模块包含了 5 个高性能的数据类型。

1. **Counter**：一个方便、快速计算的计时器工具。

```
In : import collections

In : words = ['a', 'b', 'a', 'c', 'd', 'c']

In : cnt = collections.Counter(words)
In : cnt.most_common(3)
Out: [('a', 2), ('c', 2), ('b', 1)]
```

Counter 除了可以接受多种类型的参数以及方便获得根据计数后的排序外，还有一个最重要的功能，就是可以和其他 Counter 实例做计算：

```
In : cnt2 = collections.Counter('alcdcae')
In : cnt + cnt2 # 两个计数结果组合
Out: Counter({'a': 4, 'b': 1, 'c': 3, 'd': 3, 'e': 1, 'l': 1})
In : cnt - cnt2 # cnt2相对于cnt计数结果的差集
Out: Counter({'b': 1, 'c': 1})
In : cnt & cnt2 # 两个计数结果的交集
Out: Counter({'a': 2, 'c': 1, 'd': 1})
In : cnt | cnt2 # 两个计数结果的并集
Out: Counter({'a': 2, 'b': 1, 'c': 2, 'd': 2, 'e': 1, 'l': 1})
```

2. deque: 一个双端队列，能够在队列两端添加或删除队列元素。它支持线程安全，能够有效利用内存。无论从队列的哪端入队和出队，性能都能够接近于 O(1)。

```
In : d = collections.deque('ab')
In : d
Out: deque(['a', 'b'])
In : d.append('c')
In : d
Out: deque(['a', 'b', 'c'])
In : d.appendleft('d')
In : d
Out: deque(['d', 'a', 'b', 'c'])
In : d.popleft()
Out: 'd'
In : d
Out: deque(['a', 'b', 'c'])
In : d.extend('xy')
In : d
Out: deque(['a', 'b', 'c', 'x', 'y'])
In : d.extendleft('op')
In : d
Out: deque(['p', 'o', 'a', 'b', 'c', 'x', 'y'])
```

除了列表操作，deque 还支持队列的旋转操作：

```
In : d.rotate(2)
In : d
Out: deque(['x', 'y', 'p', 'o', 'a', 'b', 'c'])
In : d.rotate(-3)
In : d
Out: deque(['o', 'a', 'b', 'c', 'x', 'y', 'p'])
```

如果 rotate 的参数 N 的值大于 0，表示将右端的 N 个元素移动到左端，否则相反。

虽然 Python 内置的数据结构 list 也支持类似的操作，但是当遇到 `pop(0)` 和 `insert(0, v)` 这样既改变了列表长度又改变其元素位置的操作时，其复杂度就变为 $O(n)$ 了。

3. `defaultdict`。`defaultdict` 简化了处理不存在的键的场景。如果不使用 `defaultdict`，对一个单词的计数要这样实现：

```
In : d = {}
```

```
In : words
```

```
Out: ['a', 'b', 'a', 'c', 'd', 'c']
```

```
In : for w in words:
```

```
....:     if w in d:
```

```
....:         d[w] += 1
```

```
....:     else:
```

```
....:         d[w] = 1
```

```
....:
```

如果使用 `defaultdict`，就不需要判断字典中是否已经存在此键：

```
In : d = collections.defaultdict(int)
```

```
In : for w in words:
```

```
....:     d[w] += 1
```

```
....:
```

`defaultdict` 参数就是值的类型，还可以使用自定义类型。下面演示一个插入后自动排序的自定义列表类型：

```
In : import bisect
```

```
In : class bisectedList(list):
```

```
....:     def insort(self, arr):
```

```
....:         bisect.insort_left(self, arr)
```

```
....:
```

```
In : d = collections.defaultdict(bisectedList)
```

```
In : d['l'].insort(1)
```

```
In : d['l'].insort(9)
```

```
In : d['l'].insort(3)
```

```
In : d['l']
```

```
Out: [1, 3, 9]
```

`defaultdict` 对于不同的数据结构都有默认值，比如 `int` 的默认值是 0：

```
In : d = collections.defaultdict(int)
```

```
In : d['a']
```

```
Out: 0
```

如果想使用其他默认值，可以借用匿名函数 `lambda` 来实现：


```
In : d = collections.defaultdict(lambda x=10: x)
In : d['a']
Out: 10
In : d['b'] = 1
In : d['b']
Out: 1
```

4. `OrderedDict`。Python 的 `dict` 结构是无序的：

```
In : d = dict([('a', 1), ('b', 2), ('c', 3)])
In : for k, v in d.items():
....:     print k, v
....:
a 1
c 3
b 2
```

在一些场景下，是必须要有顺序的，可以使用 `OrderedDict` 这个数据结构来保证字典键值对的顺序：

```
In : d = collections.OrderedDict([('a', 1), ('b', 2), ('c', 3)])
In : for k, v in d.items():
...:     print k, v
...:
a 1
b 2
c 3
```

5. `namedtuple`。`namedtuple` 能创建可以通过属性访问元素内容的扩展元组。使用 `namedtuple` 能创建更健壮、可读性更好的代码。假设不使用 `namedtuple`，查询数据库获得如下一条记录：

```
user = ('xiaoming', 23, 'beijing', '01012345678')
```

需要通过索引获取对应的内容，比如想知道用户的年龄，就得使用 `user[1]`。在大型应用中使用这种不明显的方式会加大项目的复杂度和维护成本，因为每个看到这段代码的人都得去找到 `user` 的定义，再联想 `user[1]` 是什么。使用 `namedtuple` 则让代码非常清晰：

```
In : User = collections.namedtuple('User', 'name age location phone')
In : user = User('xiaoming', 23, 'beijing', '01012345678')
In : user.age
Out: 23
In : user._asdict()
Out:
OrderedDict([('name', 'xiaoming'),
              ('age', 23),
```

```
        ('location', 'beijing'),  
        ('phone', '01012345678'))]
```

`user.age` 这个名字可以明确地告诉开发者它所表达的意义，同时还可以非常方便地把从数据库等来源获得的数据转化为 `User` 对象：

```
In : cursor = conn.cursor()  
In : cursor.execute('SELECT name, age, location, phone FROM users')  
In : for user in map(User._make, cursor.fetchall()):  
...:     print user.name, user.phone
```

Python 语法最佳实践

无论初学者还是非常有经验的工程师，或多或少都听过 Python 之禅（<https://www.python.org/dev/peps/pep-0020/>）。笔者在初学 Python 的时候也曾经通过“import this”看过它的内容，但是由于它的语句非常让人费解而没有留下太多印象。直到笔者使用 Python 4 年多之后再去看，才渐渐领会到这些语句想要传达的 Python 的设计哲学。

开始讲述本节内容之前，我们很有必要把它列出来复习一遍，并逐行解读：

优美胜于丑陋。无论什么级别的工程师都应该总是寻找更好的实现方式，如本节介绍的一些实践。除此之外，经常读一些优秀的开源项目和 Python 标准库，也可以帮你快速提升写 Python 代码的品味。

明了胜于晦涩。大概每个人都曾经在代码中炫过技。当学会了一个新的知识，笔者也曾恨不得马上应用到自己的代码里面，有时候甚至会为了使用这些技巧而写。炫技的结果就是代码晦涩难懂，其他维护者或者一段时间之后你自己再来看这段代码就会需要花费更多的时间才能理解它要表达的意图。

简洁胜于复杂。要在合适的地方使用最合适的技术，用最简洁的代码实现功能即可。

复杂胜于凌乱。如果复杂不可避免，但是仍然可以在实现功能的时候让代码结构和接口明确，各模块之间关系有层次感，利于阅读和维护。

扁平胜于嵌套。不要使用太多的嵌套，嵌套太深不利于阅读和理解。

间隔胜于紧凑。不要妄图一行代码实现功能，整段代码没有空行作为间隔，堆在一起。适量的空行使代码逻辑更清晰，应该适当地把功能的不同部分用空格分隔开。

可读性很重要。代码是给人读的，顺便让机器执行。除了让代码实现得更简洁明了，命名也很重要，下面我们会讲到一些提高可读性的原则。

特例并没有特殊到可以违背规则的地步。Python 相对于其他语言有比较多的限制，很多人懂 Python 代码之美，也有很多人觉得代码能运行就好了，至于怎么组织代码，怎么提升

运行效率，怎么写得更 Pythonic，他们并不关心。笔者也会遇到某些特例，比如一句话放在一行超过 79 个字符，拆分成 2 行又很奇怪。笔者通常为了可读性选择放在一行，但是在行尾添加“# noqa”标识，保证不让测试失败，这并没有完全遵守 PEP8。最怕的是没有意识，完全不考虑阅读和维护这段代码的其他人的感受。

虽然适用性胜于纯粹性，也不该默默地忽略错误，除非明确地忽略。之前强调了不要捕获全部错误，也不要完全忽略。下面这样的代码没有任何意义，应该尽量避免：

```
try:
    xxx
except:
    pass
```

当存在多种可能时，不要尝试去猜测，尽量找一种，最好只有一种明显的解决方案，虽然这并不如意，因为你不是 Python 之父。编程的乐趣之一是可以使用多种方式实现同样的功能，但是往往最优解只有一个。在学习 Python 的路上，这个最优解会随着你对 Python 的熟悉而改变，前提是你愿意花时间去思考当前所能使用的最好的方式是不是那个最优解，有没有更好的解决方案？

做也许好过不做，但盲目动手还不如不做。写代码之前要明确需求，考虑清楚怎么组织代码，实现的难点有哪些等，避免花了很多精力实现一半的时候才发现其实一开始的方向就有问题，甚至需要推翻重来。

如果方案很难解释，那就不是最好的方案。复杂的方案往往意味着逻辑复杂，需要花费更多的时间和资源。

如果方案容易解释，那么有可能这是一个好方案。容易解释，说明开发者思路清晰，而思路清晰在方案评审的时候也可以让其他人更容易理解，更好地评判方案的可行性。

命名空间是一种绝妙的理念，我们应当多加利用。合理利用命名空间，减少隐式的命名冲突。

Python 有一些惯用写法，Pythonic 的写法相对更简练、明确、优雅，绝大部分时候执行效率更高，而且代码越少也就越不容易出错。如下资料提供了一些写出优美的 Pythonic 代码的实践：

- sloria 整理的 Python 最佳实践（<http://bit.ly/1PwdxH5>）
- JeffPaine 整理的 Idiomatic Python（<http://bit.ly/265xmPd>）

每个工程师都应该熟练掌握上述文档。

命名

统一的命名风格可以让项目具有更高的可读性。

1. 变量、函数、方法和包名使用全小写字母和下画线的组合,如 `lower_case_with_underscores`。
2. 类和异常使用大写字母开头的单词,如 `CapWords`。
3. 受保护的或者内部方法可以使用一个下画线开头,如 `_single_leading_underscore(self, ...)`。
4. 私有方法使用两个下画线开头,如 `__double_leading_underscore(self, ...)`。
5. 常量使用全大写的字母和下画线的组合,如 `ALL_CAPS_WITH_UNDERSCORES`。
6. 尽量不要使用单个字符的变量,除非是上下文非常清晰(如在列表解析中作为中间变量)的情况。
7. 避免多余的标签。如果类或者包的名字已经包含对应的单词,那么对应的属性或者方法上就不用出现了。举个例子:

```
import audio

core = audio.AudioCore()
controller = audio.AudioController()
```

可以简化为:

```
import audio

core = audio.Core()
controller = audio.Controller()
```

8. 动词和形容词放在名词之后。如使用:

```
elements = ...
elements_active = ...
elements_defunct = ...
```

而不要把它们放在名词前面:

```
elements = ...
active_elements = ...
defunct_elements ...
```

9. 使用更容易理解的动作,比如设置属性,使用 `“user.xxx = 23”`,而不要使用 `“user.set_xxx(23)”`。

使用 join 连接字符串

常见的性能不高且代码不够优雅的使用方法是这样的：

```
In : my_list = ['X', 'y', 'p', 'J', 'F', 'r']
In : rs = ''
In : for c in my_list:
....:     rs += c + ' '
....:
In : rs = rs.strip()
In : rs
Out: 'X y p J F r'
```

正确的用法是先创建一个列表，不断地 append，最后使用 join 连接：

```
In : rs = []
In : for c in my_list:
....:     rs.append(c)
....:
In : ' '.join(rs)
Out: 'X y p J F r'
```

当然，对于这个例子直接使用 ' '.join(my_list) 就好了。当合并的元素比较少时，使用 join 方法看不出太大的效果，但是当元素多的时候，就会发现 join 的效率优势还是非常明显的。

EAFP vs LBYL

检查数据可以让程序更健壮，这就是所谓的防御性编程。数据检查有 EAFP 和 LBYL 两种编程风格。

1. LBYL: Look Before You Leap，即事先检查，通过使用 if 语句把错误输入转化成合理的用法或者返回错误信息。比如单词计数功能：

```
In : d = {}
In : words = ['a', 'b', 'c', 'a', 'a']
In : for w in words:
....:     if w not in d:
....:         d[w] = 1
....:     else:
....:         d[w] += 1
....:
```

2. EAFP: Easier to Ask Forgiveness than Permission，即不检查，出了问题由异常处理来处理。

```
In : for w in words:
...:     try:
...:         d[w] += 1
...:     except KeyError:
...:         d[w] = 1
...:
```

一般情况下，Python 程序应该倾向使用 EAFP 风格。尤其涉及原子操作时，一定要使用 EAFP 风格，比如在多线（进）程并发的时候，可能要操作的对象已经被其他线（进）程改变了。

当然，也不是都要使用 EAFP 风格，如果你预期操作在逻辑上失败的概率超过 50%，就应该使用 LBYL。

定义类的 __str__ / __repr__ 方法

自定义类的默认的 __str__ 方法的实现是无意义的：

```
In : class Board(object):
...:     def __init__(self, id, name):
...:         self.id = id
...:         self.name = name
...:
```

```
In : board = Board(1, '最新热评榜单')
```

```
In : board
```

```
Out: <__main__.Board at 0x7f735dbd6ac8>
```

好的习惯是自定义 __str__ 和 __repr__ 方法，提供有价值的输出：

```
In : class Board(object):
...:     def __init__(self, id, name):
...:         self.id = id
...:         self.name = name
...:     def __str__(self):
...:         return '({}, {})'.format(self.id, self.name)
...:     def __repr__(self):
...:         return '{}(id={}, name={})'.format(
...:             self.__class__.__name__,
...:             self.id, self.name)
...:
```

```
In : board = Board(1, '最新热评榜单')
```

```
In : board
```

```
Out: Board(id=1, name=最新热评榜单)
In : print board
(1, 最新热评榜单)
```

优美的 Python

下面将看到上述文档中提到的几个鲜为人知的例子，以及一些笔者在工作上的实践。

使用 next 获取循环中符合条件的值

通常，要从一个循环中找到符合条件的值，会使用如下方法：

```
a = -1
for i in range(1, 10):
    if not i % 4:
        a = i
        break
```

此时 a 等于 4。而更好的写法是使用 next：

```
a = next((i for i in range(1, 10) if not i % 4), -1)
```

执行调用直到某种情况结束

使用如下方式读取文件：

```
blocks = []
while True:
    block = f.read(32)
    if block == '':
        break
    blocks.append(block)
```

而更好的写法是：

```
from functools import partial
```

```
blocks = []
for block in iter(partial(f.read, 32), ''):
    blocks.append(block)
```

善用 for...else 句式

常见的寻找符合条件的序列的索引值的方法如下：

```
def find(seq, target):
    found = False
    for i, value in enumerate(seq):
        if value == target:
            found = True
            break
    if not found:
        return -1
    return i
```

更好的写法是：

```
def find(seq, target):
    for i, value in enumerate(seq):
        if value == target:
            break
    else:
        return -1
    return i
```

第二种方法不用引入 found 这个变量。

最简单的缓存实现

最常见的实现缓存的方法如下：

```
def web_lookup(url, saved={}):
    if url in saved:
        return saved[url]
    page = urllib.urlopen(url).read()
    saved[url] = page
    return page
```

这样的缓存用法比较隐晦，而且灵活性不足，可以简单地改写成装饰器风格的缓存：

```
def cache(func):
    saved = {}
    @wraps(func)
    def newfunc(*args):
        if args in saved:
            return saved[args]
        result = func(*args)
```



```
        saved[args] = result
    return result
return newfunc

@cache
def web_lookup(url):
    return urllib.urlopen(url).read()
```

从 Python 3 移植

Python 3 添加了很多 Python 2 没有的功能，可以把这些用法移植到使用 Python 2 的项目中。

partialmethod

顾名思义，`partialmethod` 是作用于类方法的 `partial` 函数：

```
def get_name(self):
    return self._name

class Cell(object):
    def __init__(self):
        self._alive = False

    @property
    def alive(self):
        return self._alive

    def set_state(self, state):
        self._alive = bool(state)
    set_alive = partialmethod(set_state, True)
    set_dead = partialmethod(set_state, False)

    get_name = partialmethod(get_name)
```

这样使用它：

```
In : cell = Cell('cell_1')
In : cell.alive
Out: False
In : cell.set_alive()
In : cell.alive
Out: True
In : cell.get_name()
```

Out: 'cell_1'

在 Python 2 中可以这样实现：

```
from functools import partial
```

```
class partialmethod(partial): # noqa
    def __get__(self, instance, owner):
        if instance is None:
            return self
        return partial(self.func, instance,
                       *(self.args or ()), **(self.keywords or {}))
```

这个使用描述符实现的版本并不是 Python 3 标准库中的实现，它只能作用于类方法，不能作用于已经被 `partialmethod` 包装的方法。举个 Python 3 的例子：

```
from functools import partialmethod
```

```
class Request(object):
    default_url = 'http://www.dongwm.com'

    def request(self, method, url, params=None, data=None):
        print('execute request: {}'.format(url))

    get = partialmethod(request, 'GET')
    post = partialmethod(request, 'POST')
    put = partialmethod(request, 'PUT')
    delete = partialmethod(request, 'DELETE')

    get_default_url = partialmethod(get, default_url)
```

使用 Python 2 版本的 `partialmethod` 不能包装这个例子中的 `get` 方法，也就是不能实现 `get_default_url` 这个方法。如果有这样的需求，就需要从 Python 3 标准库中移植代码了。

singledispatch

Python 3.4 为 `functools` 模块引入了将普通函数转换为泛型函数的工具 `singledispatch`。泛型设计是一种编程范式，泛型函数是一组实现不同类型输入，但是执行相同操作的的函数。

在 Python 2 中使用 `singledispatch` 需要安装第三方包：

```
> pip install singledispatch
```

我们知道 JSON 默认并不能序列化时间格式的值，举个例子：

```
import json
from datetime import date, datetime

class Board(object):
    def __init__(self, id, name, create_at=None):
        self.id = id
        self.name = name
        if create_at is None:
            create_at = datetime.now()
        self.create_at = create_at

    def to_dict(self):
        return {'id': self.id, 'name': self.name,
                'create_at': self.create_at}

board = Board(1, 'board_1')
```

如果想序列化 to_dict 方法的结果，就会报错：

```
In : print(json.dumps(board.to_dict()))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-2-9225d99f1e8c> in <module>()
----> 1 print(json.dumps(Board(1, 'board_1').to_dict()))
...
TypeError: datetime.datetime(2016, 6, 08, 4, 14, 29, 857598) is not JSON serializable
```

这个时候需要在 to_dict 方法中对时间格式的值做特殊处理，通常是添加一个 encoder 函数，函数代码如下：

```
def json_serial(obj):
    if isinstance(obj, datetime):
        serial = obj.isoformat()
        return serial
    TypeError(repr(obj) + ' is not JSON serializable')

json.dumps(board.to_dict(), default=json_serial)
```

特殊类型越多，这样的判断也就越多，性能越差。如果使用 singledispatch 实现，则代码如下：

```
from singledispatch import singledispatch
```

```
@singledispatch
```

```
def json_encoder(obj):
    raise TypeError(repr(obj) + ' is not JSON serializable')
```

```
@json_encoder.register(date)
@json_encoder.register(datetime)
def encode_date_time(obj):
    return obj.isoformat()
```

```
json.dumps(board.to_dict(), default=json_encoder)
```

除了转换函数，还可以转换方法：

```
from functools import update_wrapper
```

```
def methdispatch(func):
    dispatcher = singledispatch(func)

    def wrapper(*args, **kw):
        return dispatcher.dispatch(args[1].__class__)(*args, **kw)
    wrapper.register = dispatcher.register
    update_wrapper(wrapper, func)
    return wrapper
```

现在给 Board 添加两个泛型方法：

```
@methdispatch
def get(self, arg):
    return getattr(self, arg, None)

@get.register(list)
def _(self, arg):
    return [self.get(x) for x in arg]
```

现在就可以对 get 方法传入不同的参数获取不同类型的结果了：

```
In : print(board.get('name'))
board_1
In : print(board.get(['id', 'create_at']))
[1, datetime.datetime(2016, 5, 17, 4, 14, 29, 857598)]
```

suppress

Python 3 的 contextlib 模块中，suppress 通过 with 语句忽略指定的异常，Python 2 的移植版本如下：

```
from contextlib import contextmanager
```

```
@contextmanager
def suppress(*exceptions):
    try:
        yield
    except exceptions:
        pass
```

```
with suppress(OSError):
    os.remove('/no/such/file')
```

上述的删除不存在对文件的操作，所以不会抛出异常。

redirect_stdout/redirect_stderr

有时候为了调试之类的目的，希望把输出重定向到某个文件。之前的做法是：

```
with open('help.txt', 'w') as f:
    oldstdout = sys.stdout
    sys.stdout = f
    try:
        help(pow)
    finally:
        sys.stdout = oldstdout
```

Python 3 的 contextlib 提供了更优雅的方式，Python 2 的移植版本如下：

```
@contextmanager
def redirect_stdout(fileobj, std_type='stdout'):
    oldstdout = getattr(sys, std_type)
    setattr(sys, std_type, fileobj)
    try:
        yield fileobj
    finally:
        setattr(sys, std_type, oldstdout)
```

```
redirect_stderr = partial(redirect_stdout, std_type='stderr')
```

可以通过如下方式使用它们：

```
with open('help_out.txt', 'w') as out, open('help_err.txt', 'w') as err:
    with redirect_stdout(out), redirect_stderr(err):
```

```
msg = 'Test'
sys.stdout.write('(stdout) A: {!r}\n'.format(msg))
sys.stderr.write('(stderr) A: {!r}\n'.format(msg))
```

使用 CFFI/Cython 编写 Python 扩展

除了使用并发编程，还有三种方法可以提高 Python 代码的执行效率。

1. 使用 PyPy。PyPy 使用即时编译器（Just in Time compiler，简称 JIT）编译代码，对于长期执行的程序能明显提高效率，如 Web 服务。
2. 通过第三方工具让 Python 程序调用 C/C++ 代码。目前最流行的方式有以下 3 种：
 - SWIG。SWIG 可以把 C/C++ 的代码封装成 Python 库，供 Python 调用。使用 SWIG 可以有效利用脚本语言的开发效率和 C/C++ 的运行效率。
 - Boost.Python。Boost.Python 是 Boost 中的一个组件，使用它能够大大简化用 C++ 为 Python 写扩展库的步骤，提高开发效率。
 - CFFI。CFFI 实现类似 ctypes 的访问 C 库并调用其函数的功能。PyPy、cryptography、PIL 等都使用了它。
3. 改用 Cython 编写代码。

使用 CFFI

开发者只要会 C 和 Python 就可以使用 CFFI，而且大部分场景下直接从头文件或者文档拷贝声明即可。

我们先安装它：

```
> pip install cffi
```

CFFI 有 ABI 和 API 共两种模式，每种模式下又包含 in-line 和 out-of-line 这两种编译模式。in-line 表示即时编译使用，常用来做效果测试；out-of-line 表示离线编译后调用，生产环境都使用这种模式。我们在 C 标准库参考教程（<http://bit.ly/1OsuDLd>）上找到 ceil 函数来感受一下这 4 种模式。

1. ABI 的 in-line 模式。ABI 模式不需要任何 C 编译器：

```
In : from cffi import FFI
In : ffi = FFI()
In : ffi.cdef('double ceil(double x);')
In : C = ffi.dlopen(None) # 加载动态链接库，使用None表示加载C标准库
In : val1 = ffi.cast('float', 10.9)
```

```
In : C.ceil(val1)
Out: 11.0
```

其中 `cdef` 方法中的内容是直接从文档中拷贝的，但是要确保行尾有分号。

2. ABI 的 out-of-line 模式。

```
from cffi import FFI

ffi = FFI()
ffi.set_source('_abi_out', None)
ffi.cdef('double ceil(double x);')

ffi.compile()
```

执行 `ffi.compile` 方法后会生成 `_abi_out.py` 文件，接下来的操作都基于 `_abi_out.py`：

```
from _abi_out import ffi as ffi_
lib = ffi_.dlopen(None)
print lib.ceil(ffi.cast('float', 10.9))
```

3. API 的 in-line 模式。在线写 C 代码即可：

```
In : from cffi import FFI
In : ffi = FFI()
In : ffi.cdef('double ceil(double x);')
In : lib = ffi.verify('double ceil(double x);')
In : lib.ceil(10.9)
Out: 11.0
```

4. API 的 out-of-line 模式。ABI 的 out-of-line 生成的是 Python 代码，而 API 使用了编译器，会生成 `.c`、`.o` 和 `.so` 文件：

```
from cffi import FFI
ffi = FFI()

ffi.set_source(
    '_api_out',
    '''
        #include <math.h>
    ''')

ffi.cdef('double ceil(double x);')

ffi.compile()

from _api_out import lib
print lib.ceil(10.9)
```

能产生这样的区别，原因在于 `set_source` 方法的第二个参数是不是 `None`。

上面的例子都是使用 C 标准库的函数，现在演示一个完整的例子。首先创建一个头文件 (`board.h`)，文件内容主要是函数、结构声明、常量定义等：

```
typedef struct {
    int p_id;
    wchar_t *p_name;
} board_t;

board_t *create(int id, const wchar_t *name);
void board_destroy(board_t *p);
```

其中定义了一个叫作 `board_t` 的结构体，它包含 `p_id` 和 `p_name` 两个字段；还定义了创建和销毁结构体的函数。

然后创建一个源文件 (`board.c`，CFFI 编译后会生成完整的 `.c` 源文件)，`.c` 文件存放函数定义，`board.c` 中包含了创建和销毁结构体的函数：

```
#include <stdlib.h>
#include <wchar.h>

board_t *create(int id, const wchar_t *name) {
    board_t *p = malloc(sizeof(board_t));
    if (!p)
        return NULL;
    p->p_id = id;
    p->p_name = wcsdup(name);
    return p;
}

void board_destroy(board_t *p) {
    if (p->p_name)
        free(p->p_name);
    free(p);
}
```

使用 API 的 `out-of-line` 模式来创建 `_board.so` (`build_board.py`)：

```
import os

from cffi import FFI

ffi = FFI()
here = os.path.dirname(__file__)

with open(os.path.join(here, 'board.h')) as f:
```



```
header = f.read().strip()

with open(os.path.join(here, 'board.c')) as f:
    source = f.read().strip()

ffi.set_source('_board', '\n'.join([header, '', source]))
ffi.cdef(header)

ffi.compile()
```

运行之后，可以看到已经在当前目录下生成了动态链接库 `_board.so`。使用它：

```
from _board import ffi, lib

class Board(object):
    def __init__(self, id, name):
        p = lib.create(id, name)
        if p == ffi.NULL:
            raise MemoryError('Could not allocate board')

        self._p = ffi.gc(p, lib.board_destroy)

    @property
    def id(self):
        return self._p.p_id

    @property
    def name(self):
        return ffi.string(self._p.p_name)
```

这样就能使用 `Board` 类了：

```
In : from board import Board
In : board = Board(1, u'board_1')
In : board.id, board.name
Out: (1, u'board_1')
```

使用 Cython

Cython 在本质上是包含 C 数据类型的 Python，几乎所有 Python 代码都是合法的 Cython 代码，Cython 能够把稍加修改的 Python 代码编译成 C，速度却能提升几倍到几百倍，这是并发程序很难达到的。

我们先安装它：

```
> pip install Cython
```

编辑距离 (Edit Distance)，又称 Levenshtein 距离，是指两个字符串之间由一个转成另一个所需的最少编辑操作次数。编辑距离越小，两个字符串的相似度越大。它也是字符串模糊匹配库 FuzzyWuzzy 的依赖。本节我们将对比纯 Python、只使用 Cython 编译、使用 Cython 语法编写这三种方式在效率上的提升。

我们从维基百科找到一个 Levenshtein 距离的 Python 实现 (levenshtein_p.py)：

```
def levenshtein(s, t):
    if s == t:
        return 0
    elif len(s) == 0:
        return len(t)
    elif len(t) == 0:
        return len(s)
    v0 = [None] * (len(t) + 1)
    v1 = [None] * (len(t) + 1)
    for i in range(len(v0)):
        v0[i] = i
    for i in range(len(s)):
        v1[0] = i + 1
        for j in range(len(t)):
            cost = 0 if s[i] == t[j] else 1
            v1[j + 1] = min(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost)
        for j in range(len(v0)):
            v0[j] = v1[j]

    return v1[len(t)]
```

不做任何修改，将上述代码拷贝出来，命名为 levenshtein_c.pyx，再创建一个 setup.py 文件编译它：

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name='levenshtein_c',
    ext_modules=cythonize('levenshtein_c.pyx'),
)
```

生成动态链接库：

```
> python setup_c.py build_ext --inplace
```

生成的动态链接库是 `levenshtein_c.so`。

最后再基于 `levenshtein_p.py`，创建 `levenshtein_cy.pyx`，使用 Cython 语法修改它：

```
def levenshtein(char * s, char * t):
    cdef int i, j, cost, rs
    cdef list v0, v1
    ...
```

省略的部分没有变动。可以看到，我们只是声明了参数和函数内用到的变量的类型。接下来也要通过 `setup.py` 编译它，生成 `levenshtein.so`。其中 `cdef` 用来声明 C 变量的类型。

现在对比一下三种方式生成的模块的效率：

```
In : import levenshtein_p
In : import levenshtein_c
In : import levenshtein_cy

In : timeit -n 100 levenshtein_p.levenshtein(s1, s2)
100 loops, best of 3: 3.76 ms per loop
In : timeit -n 100 levenshtein_c.levenshtein(s1, s2)
100 loops, best of 3: 1.7 ms per loop
In : timeit -n 100 levenshtein_cy.levenshtein(s1, s2)
100 loops, best of 3: 619 µs per loop
```

可以看到，`levenshtein_c` 只是通过 Cython 把代码转成 C 就让效率提升了 2 倍多，而 `levenshtein_cy` 只是对 `levenshtein` 函数简单添加一些声明，并没有做更多的优化，就可以比 Python 版本的效率高 6 倍。

再看一个使用 C 标准库中的 `math.ceil` 的例子：

```
cdef extern from "math.h":
    double ceil(double x)
```

```
cdef double f(double x):
    return ceil(x)
```

```
cpdef double f2(double x):
    return f(x)
```

`extern` 关键词引用的 `math.h` 文件中的定义，除了 `cdef` 声明 C 函数外，还出现了 `cpdef`，它是一个既能让 C 也能让 Python 调用的方式，编译之后使用一下就知道区别了：

```
In : import ceil
In : ceil.f2(10.9)
```

```

Out: 11.0
In : ceil.f(10.9)
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-3-f46aedbc5205> in <module>()
----> 1 ceil.f(10.9)
AttributeError: 'module' object has no attribute 'f'

```

使用 `cdef` 声明的函数 `f` 不是模块的一部分，但是它可以被 `f2` 函数调用。

上述例子中，声明 `ceil` 这样的常用函数是不需要找 `math.h` 头的，`Cython` 已经自带了。可以使用如下方法直接调用：

```

from libc.math cimport ceil

cpdef double f2(double x):
    return ceil(x)

```

其他可用的函数声明可以查看 `Cython/Includes` (<http://bit.ly/1XYmUrH>) 目录下相关的后缀名为 `.pxd` 的文件。

嵌入模式

除了通过动态链接库作为模块被引用，还可以借用 `Cython` 的嵌入（`Embed`）模式生成可执行的二进制程序。比如下面的小程序：

```

import sys

name = sys.argv[1] if len(sys.argv) == 2 else 'World'
print 'Hello {}'.format(name)

```

执行如下两步即可：

```

> cython --embed -o hello.c hello.py
> gcc -Os -I /usr/include/python2.7 -o hello hello.c -lpthon2.7 -lpthread -lm -lutil -ldl

```

现在可以执行 `hello` 了：

```

> ./hello
Hello World
> ./hello xiaoming
Hello xiaoming

```

进阶篇：使用 PyObjC 发送通知

越来越多的开发者选择 Mac 作为个人电脑，并用它来完成开发工作。OS X 平台上有一些独有的 Python 标准库和第三方库，它们很有用但是却鲜为人知。

当邮件客户端收到新邮件或在日历中出现新预约时，开发者会收到通知。这种通知机制也广泛应用在第三方团队协作工具（如 Slack）软件上。事实上，我们还能把通知机制应用于日常开发中，比如一个操作耗时较长，我们希望尽早了解到操作的反馈以便继续进行其他步骤，而不必一直在屏幕上等待它完成，可以切换屏幕，或者让它后台运行，我们去做其他的工作，待它完成后给我们发一个通知就好了。

Cocoa 是苹果公司为 Mac OS X 所创建的原生面向对象的编程环境，Cocoa 应用一般使用 Objective-C 开发，Python 开发者可以使用 PyObjC 这样的桥接技术编写 Cocoa 应用。Mac 自带的 Python 版本已经内置了 PyObjC，直接使用即可。

我们来实现这个发通知的应用：

```
import Foundation
from AppKit import NSImage
import objc

NSUserNotification = objc.lookUpClass('NSUserNotification')
NSUserNotificationCenter = objc.lookUpClass('NSUserNotificationCenter')

ICON_PATH = '~/web_develop/chapter14/section5'

def notify(title, subtitle, info_text, delay=0, sound=False, userInfo={},
           is_error=False):
    icon = NSImage.alloc().initWithFile_(
        os.path.join(ICON_PATH, 'douban.png'))

    notification = NSUserNotification.alloc().init()
    notification.setTitle_(title)
    notification.setSubtitle_(subtitle)
    notification.setInformativeText_(info_text)
    notification.setUserInfo_(userInfo)
    notification.setIdentityImage_(icon)
    if is_error:
        error_image = NSImage.alloc().initWithFile_(
            os.path.join(ICON_PATH, 'error.png'))
        notification.setContentImage_(error_image)
    if sound:
        notification.setSoundName_('NSUserNotificationDefaultSoundName')
```

```
notification.setDeliveryDate_(
    Foundation.NSDate.dateWithTimeInterval_sinceDate_(
        delay, Foundation.NSDate.date()))
NSUserNotificationCenter.defaultUserNotificationCenter(
).scheduleNotification_(notification)
```

`setTitle_` 方法设置标题; `setSubtitle_` 方法设置子标题; `setInformativeText_` 设置内容; `set_identityImage_` 会添加自定义图标; `notify` 接受 `sound` 参数, 如果为 `True`, 在发通知的时候会带系统默认声音。发送的通知效果如图 14.1 所示。



图 14.1 发送通知的效果

如果这是一个错误类型的通知 (`is_error=True`), 会通过 `setContentImage_` 方法额外地在通知文本右侧添加一张错误的图片, 效果如图 14.2 所示。

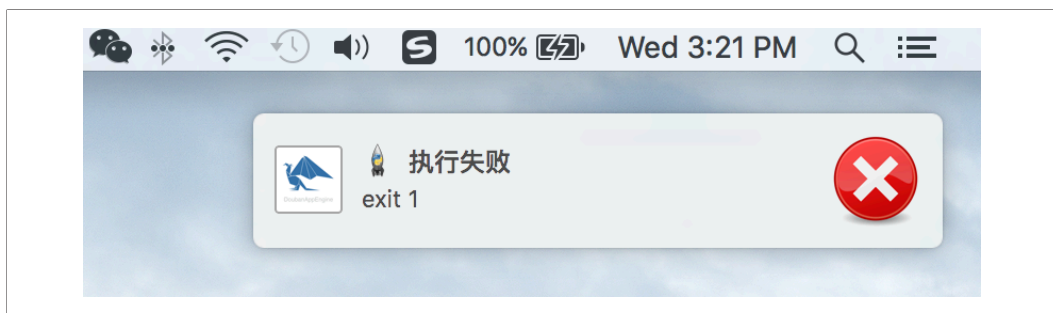


图 14.2 错误类型的通知

这是一个命令行的程序, 应该使用命令行选项与参数的解析器模块 `argparse`, 它在 Python 2.7 的时候进入标准库, 用来替代 `optparse` 模块。我们来编写发送通知的命令行接口:

```
unicode = lambda s: s.decode('utf-8')
parser = argparse.ArgumentParser()
```

```

description='Send a custom notification on OS X.')
parser.add_argument('-t', '--title', help='title of notification',
                    default='', type=unicode)
parser.add_argument('-s', '--subtitle', help='subtitle of notification',
                    default='', type=unicode)
parser.add_argument('-m', '--message', help='message of notification',
                    default='', type=unicode)
parser.add_argument('--sound', help='include audible alert',
                    action='store_true', default=True)
args = parser.parse_args()
notify(args.title, args.subtitle,
       args.message, sound=args.sound)

```

notify 需要接收 unicode 类型的值，但是 argparse 的解析结果是字符串，argparse 可以灵活地支持解析参数的值的类型转换，所以使用了 “unicode = lambda s: s.decode('utf-8')” 之类的语句就可以用 “type=unicode”，这样解析的结果就是 unicode 类型的值了。

除了发送通知，还需要把 notification.py 封装成系统命令，让发通知智能一些。举个例子，现在要执行：

```
> python runscript.py -c 4
```

封装之后就成为了：

```
> notify python runscript.py -c 4
```

这个 notify 命令接收要执行的命令及参数，通过 subprocess.call 的返回值判断执行是否成功，来决定通知的标题、类型（是否是错误类型的通知），通知内容为执行的命令。再者还要考虑发送通知的脚本的位置，因为可能开发调试是在虚拟机或者远程服务器上进行的，这就需把消息传回到 Mac 电脑上。实现这样的功能的方法很多，比如在 Mac 搭建 Web 服务，RPC 服务或者使用 SSH 服务。本节将展示在虚拟机里面通过 ssh 连接 Mac 宿主机发通知的方式：

```

#!/usr/bin/python
# coding=utf-8
import sys
import subprocess

def notify(title, msg, is_error=False):
    cmd = ('ssh user@host '
          '"python /Users/dongweiming/bin/notify.py '
          '-t \'{}\ ' -m \'{}\ '").format(title, msg)
    if is_error:
        cmd += ' --error'

```

```
subprocess.call(cmd, shell=True)

def main():
    cmd = ' '.join(sys.argv[1:])
    retcode = subprocess.call(cmd, shell=True)
    if len(cmd) > 33:
        cmd = cmd[:30] + '...'
    if retcode:
        title = '执行失败'
        is_error = True
    else:
        title = '执行成功'
        is_error = False
    notify(title, cmd, is_error=is_error)

if __name__ == '__main__':
    main()
```

把这个脚本放在 \$PATH 变量中指定的一个目录下，并添加执行权限：

```
> mkdir ~/bin
> cp chapter14/section5/notify ~/bin
> chmod +x ~/bin/notify
> export PATH=$PATH:~/bin
```

\$PATH 变量决定了 Shell 将到哪些目录中寻找命令或程序。现在不必输入这个命令的完整路径，直接输入 notify 就可以了。

第 15 章

Web 开发项目实践

本章主要是 Web 项目经验的总结，包含如下内容：

- 介绍笔者的 Web 项目开发流程和经验。
- 介绍开源的代码质量保证工具，以及豆瓣的一些质量保证实践。
- 使用 AST 对真实的业务逻辑做静态检查，实现业务流程的检查。
- 谈谈代码评审的意义和实际经验。

Web 项目经验总结

开发流程

一个产品团队里面主要有如下几种角色。

1. 产品经理（PM）：规划要做的功能，整理需求，跟进设计和开发的进度，最后验收。
2. 设计师：根据需求设计 Web 交互效果，与 PM 和工程师讨论并修改设计。
3. 工程师（Web 开发）：设计未完成前可以做一些前提准备，等设计图出来之后，工程师按照敲定的设计实现对应的页面效果和后端逻辑。
4. 测试：在开发基本结束的时候介入，验证功能。但并不是所有团队都有测试这个角色。
5. 运营：负责日常网站的维护，反馈问题，也会提需求。

以下是笔者在开发中总结的一些经验，这里分享给大家。

确定需求后再做。很多人一拿到需求，就开始先入为主地按自己的理解去做，最后的结果往往比较尴尬。笔者建议拿到需求后不要马上开始写代码，而是不断地找需求方确认，以确保对需求充分理解。

先验证那些可能实现不了的想法。拿到需求应该先做整体规划，先验证那些可能实现不了的想法，然后再去做那些只要花时间就能完成的开发工作。

优先级管理。工程师大多有完美主义情结，但是往往任务紧事情多，一定要分清优先级，对于手头的事情有整体的把控。尤其是上升期的产品，往往新增任务的速度大于处理速度，这时候就要缓一缓，如果一件事很急，需求方会不断向你询问进度。一定要明确事情的优先级，否则就把优先级不明确的事情放一放。笔者的经验是有相当一部分需求其实是需求方临时想起来的，很快在后来在考虑中发现漏洞而自己主动放弃了。

及时反馈。不要等到事情做完的时候才反馈。整个开发过程都应该是交互的、不断迭代的工程，及时获得反馈让自己不会走得太偏，也能让需求方不断了解进展情况，甚至能给开发者提供一些额外的帮助。假如由于未预期的问题或者其他外因造成不能按期完成，最好在离约定的时间半程之前反馈。

合理的项目人员配比。一般而言，实现一个小功能，一个设计师加一个工程师足矣，甚至不需要设计师；而对于比较大的项目，则需要一到两个设计师加两三个工程师。

尽量不要发太大的 PR。把安排的任务分解为功能逐个提交 PR，通常同时进行多个功能的开发，不推荐一次性提交大量代码。这会给做代码评审的同事带来负担，代码评审持续时间更长且效果会受到影响。和其他同事一起合作完成项目，非常容易出现代码冲突，以致后期需要花费大量时间和精力处理冲突。

使用合理的项目结构

无论是从零开始做一个新的项目还是对现有的架构进行优化，都是有一定的原则可遵循的。首先是项目的结构应该保持简单，有以下原则：

1. 层次不可太深，否则会造成代码分散，业务代码维护不方便。建议目录深度不超过 4 级。
2. 层次也不能太平，否则造成模块/目录数量太多、太臃肿。尽量使用目录代替模块文件的方式。
3. 善用蓝图（Blueprint）。把应用分解为一个蓝图的集合，可以方便地对这个蓝图注册或者注释，达到良好的可扩展性。

4. 大型网站不是一天设计出来，项目结构也不是一蹴而就的，在业务发展的进程中需要不间断地进行代码结构的重构工作。

关注代码复杂度

mccabe 内置于 Flake8 中，旨在检查代码复杂度。它是循环复杂度（https://en.wikipedia.org/wiki/Cyclomatic_complexity）的 Python 实现，它认为若一个模块的循环复杂度超过 10，就需要对其进行拆分。可以对项目的代码复杂度指标进行考核。

Flake8 中默认设置的复杂度检查阈值是-1，需要设置成大于 0 的值才可以启用复杂度检查：

```
> flake8 --max-complexity 3 chapter3/section7/app.py
chapter3/section7/app.py:262:1: C901 'index' is too complex (4)
chapter3/section7/app.py:321:1: C901 'preview' is too complex (4)
> flake8 --max-complexity 10 chapter3/section7/app.py
```

阈值的大小和业务复杂度、代码质量、历史遗留等因素有关，不同项目可以使用不同的阈值，但是复杂度越高、开发和维护的成本也就越高，所以阈值不宜太大。

一个函数或者一个类代码不应该过长，建议不超过显示器一屏。如果太长就应该拆分。

一个代码文件的总行数（包含空行，注释行等）也不应该过长，尤其用堆代码的方式会使代码文件越来越长。建议单文件不超过 2000 行，否则也应该做拆分。

代码质量保证工具

Python 主要的代码质量保证工具有如下 6 种。

1. Pycodestyle（<https://github.com/PyCQA/pycodestyle>）：它是一个针对 PEP8 实现的检查工具，其实就是原来的 pep8，关于其改名的讨论见 Issues 466（<https://github.com/PyCQA/pycodestyle/issues/466>）。它可以告诉你代码中哪些部分没有遵守 PEP8 的要求，并为每个问题给出其错误码。如果违反了必须遵守的规范，就会报出 E 开头的错误码；如果是其他的小问题，则会报 W 开头的错误码。
2. Pylint：一个 Python 代码风格的检查工具，它甚至能检查变量命名是否符合编码规范，声明的接口是否被真正的实现等。它对 Python 的要求近乎于苛刻，很难有项目能满足它的质量要求，但由于其可定制性非常好，使用者通常会选择忽略掉一些不希望检查的错误类型。
3. PyChecker：一个对 Python 源代码进行语法检查的工具，它不检查代码规范，只执行代码，看看是否报错，所以它能检测出一些编译期间才出现的错误，比如拼写错误、在赋值之前就使用了变量、使用的接口参数和接口定义不一致、重复定义同名

的函数/类等问题。由于它真正地执行代码，既不安全，效率也很低下，现在已经被下面说的 Pyflakes 取代了。

4. Pyflakes: 它的功能和 PyChecker 相似，但是没有真的像 PyChecker 那样去先执行再检查，所以很安全，也快了很多。一般不直接使用它，而使用集成了它的 Flake8 这个库。
5. Flake8: 现在最流行的代码检查工具。它其实包含了 Pyflakes、Pycodestyle 和 McCabe。Flake8 还有非常好的插件机制，方便根据团队习惯自定义插件。
6. autopep8: 顾名思义，它是自动地把代码格式化为符合 PEP8 标准的工具。它只支持大部分的 PEP 8 规范，但是有些 PEP 8 问题还需要人工介入。平时笔者也经常用它来处理遗留代码，甚至在笔者的个人编辑器里面集成了它 (<http://bit.ly/23e1lCx>): 原理是在保存文件前会先自动格式化成符合 pep8 要求的代码之后再保存。

Pycodestyle 对中文缩进的处理

使用 Pycodestyle 作为代码检查工具的团队可能遇到过这样的问题，对于下面的代码：

```
> cat ascii.py
story_tags = [
    (('xxxxxx', 38), [('xxxx', 39),
                      ('xxxx', 40)]),
]
> cat non-ascii.py
story_tags = [
    (('数码产品', 38), [('手机', 39),
                      ('配件', 40)])
]
```

用 Pycodestyle 执行检查的结果却是不同的：

```
> pycodestyle ascii.py

> pycodestyle non-ascii.py
non_ascii.py:4:25: E128 continuation line under-indented for visual indent
```

原因是这样的：

```
In : len('xxxx')
Out: 4
In : len('数码')
Out: 6
```

对编辑器和终端来说，屏幕的占位的长度是一致的，但是计算出的长度是不一致的，造成 Pycodestyle 误判。

Pycodestyle 维护者们不想接受笔者修改这个问题的 PR，因为增加了很多的代码和复杂度。具体的讨论见 `pycodestyle#345` (<https://github.com/PyCQA/pycodestyle/pull/345>)。但对我们来说，这个问题无法绕过，只能迂回解决：写代码的时候设置合理的换行，不要让下一行的缩进起始位的左面出现这种非 ASCII 字符就可以了。以下是两种符合要求的风格：

```
story_tags = [  
    (  
        ('数码产品', 38),  
        [('手机', 39),  
         ('配件', 40)]  
    )  
]
```

```
story_tags = [  
    (('数码产品', 38),  
     [('手机', 39),  
      ('配件', 40)])  
]
```

Flake8

笔者所在团队都是强制要求项目通过 Flake8 检查的。当有历史遗留代码或者不得不做一些硬编码（比如项目中有 Thrift 生成的代码，生成的代码不符合 Python 编码规范）等情况下，可以适当放弃对一些文件或者类型的检查。在每个项目的代码库下一般都会有一个 `setup.cfg` 文件，其中存放了 Flake8 的设置：

```
[flake8]  
exclude = main.py,tests/*,venv/*,*/*model/*_client  
ignore = F403,E265,F812,E402,E731,W503  
max-complexity = 20
```

对各选项的说明如下。

- `exclude`：不会对符合 “`main.py,tests/*,venv/*,*/*model/*_client`” 规则的文件进行检查。应该尽量减少这样的文件。
- `ignore`：会忽略 F403、E265、F812、E402、E731、W503 这几种类型错误，但也应该尽量减少这种忽略的类型。
- `max-complexity`：表示接受的代码复杂度的最大值。因为代码有历史遗留问题，所以放宽了这个值。

Pylint

代码质量检查通常在 CI 期间执行，如果检测未通过，返回值不为 0，就会让本次测试失败。之前提到了使用 Flake8 做基本的代码规范、复杂度等检查。其实还远远不够，Python 这种动态语言运行期才会做类型检查。对于参数数量不匹配，方法没有加 self，循环引用，写了 import 语句但由于包名写错了事实上无法引用等 Flake8 无法检查到的问题，可以使用 Pylint 来做这样的静态检查。

Pylint 提供了非常多的检查类型，但是太苛刻，使用时应该只验证想要验证的类型。

代码质量检查程序如果不想调用 subprocess 执行命令获取返回值，可以使用如下方式集成 Pylint (pylint_example.py)：

```
import os
import sys

from pylint import lint

if len(sys.argv) > 1:
    FILES = sys.argv[1:]
else:
    FILES = []
    for dirpath, dirnames, filenames in os.walk(os.getcwd()):
        FILES.extend(
            os.path.join(dirpath, filename)
            for filename in filenames
            if filename.endswith('.py')
        )

MESSAGES = ['C0202', 'E0102', 'E0211', 'E0213', 'E1120', 'E1121',
            'E1123', 'W0613', 'R0401', 'R0801']

args = [
    '--reports=n',
    '--disable=all',
    '--msg-template="{path}:{line}: [{msg_id}, {obj}] {msg}"',
    '--enable={}'.format(', '.join(MESSAGES))
]

args.extend(FILES)
sys.exit(lint.Run(args))
```

args 的参数含义如下：

- `--reports=n` 表示不生成报告。
- `--disable=all` 表示先全部禁止，再使用 `--enable=C0202,E0102...` 开启要检测的类型的方式。
- Pylint 的默认输出没有完整的错误序列号，所以通过 `--msg-template` 重新定制了错误模板。
- 只检测 MESSAGES 列表中列出的类型。

其他代码质量保证工具

Gandalf

Gandalf (<http://bit.ly/21osKjT>) 是笔者开源的一个基于 `linty_fresh` 实现的 GitHub Webhook 服务。使用 `asyncio`、`rq`、`Flask` 等技术。在提 PR 之后会计算本次修改是否符合 Flake8 检查，如果有不符合项会在 PR 的对应行上添加评论，指出错误。详细的使用方法和效果可以参考笔者的博客 (<http://bit.ly/1UQt5d6>)。

Pre-commit

Pre-commit 是一个质量非常高，代码测试覆盖率达到 99% 以上的项目。它通过定义 Git 的 `pre-commit/pre-push` 钩子，在本地提交 `commit` 或者 `push` 时，对一些指标进行检查。如果不符合，则不允许提交成功。本地检查可以有效地提前发现一些代码质量问题，也减轻了线上持续集成系统的负担。

使用 AST 做静态检查

CPython 的编译过程中会先根据源代码建立抽象语法树 (Abstract Syntax Tree, AST)，最后编译为代码对象。Python 标准库中已经自带了 AST 模块，名为 `ast`。它通常用来做静态文件的检查和修改代码的执行效果：

```
In : import ast
In : node = ast.parse('2 + 6', mode='eval')
In : eval(compile(node, '<string>', 'eval'))
Out: 8
In : node.body.op = ast.Sub()
In : eval(compile(node, '<string>', 'eval'))
Out: -4
```

通过修改 body 的 op 属性，让相加的操作变成了相减。这就是 AST 的威力。可以把代码想象成一棵树，缩进的代码块是一棵子树，每一条语句都是 ast 对应类型的实例。

修改语法树节点的更好的方法是通过 NodeTransformer 类完成：

```
class RewriteAddToSub(ast.NodeTransformer):

    def visit_Add(self, node):
        node = ast.Sub()
        return node

node = ast.parse('2 + 6', mode='eval')
node = RewriteAddToSub().visit(node)
print eval(compile(x, '<string>', 'eval')) # 等于-4
```



如果只是浏览对应节点可以使用 ast.NodeVisitor。

Pylint、PyChecker 和 PyFlakes 都是基于 AST 做的静态检查。除了检查语法，同样可以检查业务流程，也就是检验代码实现中的漏洞。举个例子，Web 应用的模型不应该直接读数据库，需要添加缓存。新增、更新以及删除等操作都要清理缓存。有时候开发者忘记在删除的时候清除缓存了，这是非语法的错误，除非极为熟悉逻辑甚至之前踩过这样的坑，才能发现此类问题。这种隐藏的 bug 会给故障排除带来很大的难度。下面是一个正确的例子：

```
class MovieOrder(object):
    MC_KEY = 'chapter15:section2:movie_roder:%s'

    def __init__(self, id, type_id, order_id, price):
        self.id = id
        self.type_id = type_id
        self.price = price

    @classmethod
    @cache(MC_KEY % '{id}')
    def get(cls, id):
        sql = ('select id, type_id, order_id, price from movie_order '
              'where id=%s')
        rs = store.execute(sql, id)
        return cls(*rs[0]) if rs else ''

    @classmethod
    def delete(cls, id):
        sql = 'delete from movie_order where id=%s'
        try:
```



```

        store.execute(sql, id)
        store.commit()
    except IntegrityError:
        store.rollback()
        return False

    cls.clear_mc(id)
    return True

def update_price(self, price):
    sql = 'update movie_order set price=%s where id=%s'
    updated = store.execute(sql, (price, self.id))
    if updated:
        store.commit()
        mc_delete(self.MC_KEY % self.id)
    return updated

@classmethod
def clear_mc(cls, id):
    mc_delete(cls.MC_KEY % id)

```

上面的模型有如下细节需要注意：

- 根据 id 从数据库获取条目之后，会缓存这个结果以备在未过期范围内重复使用。
- 把清理缓存独立成一个方法，执行 delete 方法时除了删除数据库中对应条目还要清理对应的缓存。
- 为了演示，update_price 中直接使用了 mc_delete 函数，而没有使用 clear_mc 方法。

如果我们用 AST，要怎么做呢？

1. 如果类的方法中带有 SELECT 的 SQL 语句，那么它就应该被缓存，也就是需要使用 cache 这个装饰器。
2. 如果类的方法中带有 UPDATE、DELETE 的 SQL 语句，就需要在执行成功后清理缓存。
3. 清理缓存可以直接使用 mc_delete 这个方法，参数的文本需要包含 MC_KEY。也可以把缓存清理放在一个名为 clear_mc（约定）的方法中去做。

一开始先定义了 4 个变量：

```

USE_CLEAR_METHOD_LIST = []
NEED_CLEAR_LIST = []
IGNORE_LIST = []
MSG_MAP = {}

```

设计得如此复杂主要是因为第二个需求，清理缓存既可以放在 `clear_mc` 这个方法中，也可以直接在方法内部使用 `mc_delete`。在一个方法中发现没有 `mc_delete`，但是确实有 `clear_mc`，可是这不代表 `clear_mc` 方法正确地清理了缓存，因为无法强制要求开发者一定要把 `clear_mc` 写在最上面，最先被遍历到，此时可能还没有遍历到 `clear_mc` 方法，所以要用三个列表作为缓存。

1. `USE_CLEAR_METHOD_LIST`：如果一个方法中使用 `clear_mc`，会把这个方法名保存下来。
2. `NEED_CLEAR_LIST`：如果一个方法中使用了更新或者删除的 SQL 语句，它就应该清理缓存，要在 `NEED_CLEAR_LIST` 中添加这个方法名。
3. `IGNORE_LIST`：即便 `NEED_CLEAR_LIST` 中包含了 `IGNORE_LIST` 中的方法名，程序也认为不应该报错，因为在方法内正确地使用了 `mc_delete`，我们就没有必要继续验证，直接忽略。
4. `MSG_MAP`：MSG_MAP 用来存放方法名和对应的打印错误信息的偏函数。

基于上述原则，先看一下遍历抽象语法树的方法：

```
with open(filename) as f:
    tree = ast.parse(f.read())
for stmt in ast.walk(tree):
    if not isinstance(stmt, ast.ClassDef):
        continue

    has_clear_method = False

    for body_item in stmt.body:
        if not isinstance(body_item, ast.FunctionDef):
            continue
        item_name = body_item.name
```

其中 `has_clear_method` 为 `True`，则表示存在 `clear_mc` 这个方法，且方法中清理了 `MC_KEY` 的缓存。

使用 `ast.walk` 遍历语法树，因为我们只需要找到 `ClassDef` 类型的节点。每个子树如果有节点就会有 `body` 属性，我们遍历 `stmt.body` 也会忽略非 `ast.FunctionDef` 实例的节点。

`body_item` 记录了这个方法的位置，遍历完之后假如不符合要求，要打印对应的行数、偏移量等信息，所以需要预先创建一个偏函数：

```
def msg(filename, lineno, offset, msg):
    print '{}: {}: {} {}'.format(filename, lineno, offset, msg)
```

```
msg_p = partial(msg, filename, body_item.lineno,
                body_item.col_offset)
```

msg_p 现在只需要接受 msg 这个参数了。它会存放进 MSG_MAP，在最后检查的时候取出来执行。

遍历 body_item.body 就获得了方法内第一级节点。下面的检查都以 item 为单位，我们看一下使用了 select 的方法是否用了 cache 这个装饰器：

```
for item in body_item.body:
    if isinstance(item, ast.Assign):
        value = item.value
        if isinstance(value, ast.Str):
            value = item.value

            if 'select' in value.s:
                for deco in body_item.decorator_list:
                    if isinstance(deco, ast.Call):
                        if deco.func.id == 'cache':
                            break
            else:
                msg_p('Need `cache` decorator!')
```

本例的 SQL 都是一个赋值语句，所以只要找 ast.Assign 类型的节点，然后通过 ast.str 获得赋值的值的源码即可。如果包含 select 就是符合条件的方法，这个时候通过 body_item.decorator_list 找到对应方法的装饰器列表；如果 for 循环结束也没有找到 cache 这个装饰器，说明没有添加 cache 装饰器，然后通过 msg_p 把对应的错误打印出来。

value.s 就是 SQL 语句的值，除了看是不是包含 select，还能通过是否包含 delete 和 update 这两个字符串来确定是否需要清除缓存：

```
if 'select' in value.s:
    ...
elif any(op in value.s for op in ('delete', 'update')):
    NEED_CLEAR_LIST.append(item_name)
    MSG_MAP[item_name] = msg_p
```

再看一下，确认方法中使用 clear_mc 以及确认 clear_mc 方法正确性的逻辑：

```
if isinstance(item, ast.Assign):
    ...
elif isinstance(item, ast.Expr) and isinstance(item.value, ast.Call):
    func = item.value.func
    if isinstance(func, ast.Attribute):
        if func.attr == 'clear_mc':
            USE_CLEAR_METHOD_LIST.append(item_name)
```

```

else:
    if func.id != 'mc_delete':
        continue
    for arg in item.value.args:
        if not isinstance(arg, ast.BinOp):
            continue
        if 'MC_KEY' in arg.left.attr:
            if body_item.name == 'clear_mc':
                has_clear_method = True

```

上述代码做了两件事：

- 如果一个方法中调用了 `clear_mc`，则将此方法放入 `USE_CLEAR_METHOD_LIST` 中。
- 如果有 `clear_mc` 方法以及方法中正确清理了包含 `MC_KEY` 内容的键的缓存，则把 `has_clear_method` 设置为 `True`。

接下来检查方法内是否使用了正确的 `mc_delete`：

```

if isinstance(item, ast.Assign):
    ...
elif isinstance(item, ast.If):
    for if_item in item.body:
        if isinstance(if_item.value, ast.Call):
            func = if_item.value.func
            if isinstance(func, ast.Name):
                if func.id != 'mc_delete':
                    continue
            for arg in if_item.value.args:
                if not isinstance(arg, ast.BinOp):
                    continue
                if 'MC_KEY' in arg.left.attr:
                    IGNORE_LIST.append(item_name)

```

由于封装的 `store.execute` 方法会带有表示执行操作是否成功的返回值，所以这种清理缓存的操作一般在 `if` 语句内部，判断方式和上面确认 `clear_mc` 方法中清理缓存的逻辑基本类似。

最后在遍历完整个类之后开始验证，验证的原则是：

1. `NEED_CLEAR_LIST` 列表中每个元素都是需要被确认的方法名，遍历它。
2. 如果方法名在 `IGNORE_LIST` 列表中，直接忽略。
3. 如果有 `clear_mc` 方法且使用正确，并且要验证的方法在 `USE_CLEAR_METHOD_LIST` 中，则使用方式是正确的，否则都是错误的。
4. 通过错误的方法就可以在 `MSG_MAP` 中找到对应的函数打印错误信息。

```

for method_name in NEED_CLEAR_LIST:

```

```
if method_name not in IGNORE_LIST:
    if not (has_clear_method and method_name in
            USE_CLEAR_METHOD_LIST):
        MSG_MAP[method_name]('Need clear mc!')
```

其他静态检查工具

bandit

bandit (<https://github.com/openstack/bandit>) 是 OpenStack 安全小组开发的基于 AST 的静态分析工具，其中包含了很多有用的检查，尤其是安全方面的。它可以作为 Pylint 的补充。列举如下几个错误类型：

1. `try_except_pass` (B110) / `try_except_continue` (B112)：错误的代码风格。
2. `flask_debug_true` (B201)：开发者可能会在线上开启 DEBUG 模式，这非常不安全。
3. `subprocess_popen_with_shell_equals_true` (B602)：使用 `shell=True` 的方式执行系统命令不安全。
4. `hardcoded_sql_expressions` (B608)：工作中会发现有非常多不规范的拼 SQL 的代码，它们都有安全问题。
5. `eval` (B307)：验证是否使用了 `eval`。如果必须要用 `eval`，可以使用安全的 `ast.literal_eval` 替代。

hacking

hacking (<https://github.com/openstack-dev/hacking>) 是 OpenStack 开发者使用的风格检查工具，而且其中的每个检查都是一个 Flake8 插件。这个库主要使用 `tokenize` 和正则表达式对代码做检查，`tokenize` 是标准库中的词法解析模块，它可以把源码字符串拆分成单词。

列举几个常用检查类型：

- TODO 格式。
- 文档字符串格式。
- 异常处理的格式。和 bandit 一样，不允许 `try_except_pass` 的编码风格。
- 要求 `import` 按照字母顺序排列。

zhang-shasha

虽然 Pylint 也支持相似度的检查，但是如果要检查拷贝之后略加修改的重复代码，效果很差。zhang-shasha (<https://github.com/timtadh/zhang-shasha>) 是基于树的编辑距离算法，通过 ast 解析抽象语法树，可以基于它增强代码相似度的检查。

编写 Flake8 扩展

Flake8 设计了非常好的插件系统，现有的常用插件有如下几种。

- flake8-immediate: 不用等处理全部完成，有错误就打印出来。
- flake8-print: 不允许代码中出现 print 语句。
- pep8-naming: 检查命名风格，比如类名是否是首字母大写的 CapWords 风格，函数名是否是全小写，类方法第一个参数是 cls 等。
- flake8-quotes: 强制要求使用单引号，而不能用双引号；或者相反。本书例子使用单引号风格。

现在把检查缓存使用的功能也改写成一个 Flake8 扩展。首先创建一个目录，然后添加 flake8_mc.py 文件，定义一个类：

```
class McChecker(object):
    name = 'flake8-mc'
    version = '0.1.0'

    def __init__(self, tree, filename):
        self.tree = tree

    def run(self):
        return main(self.tree, self)
```

需要定义 name 和 version 这两个属性。在初始化的时候需要添加 tree 和 filename 参数，由于我们使用 AST，直接操作这个生成的语法树就好了，filename 只占位不使用。

main 函数的内容和上面的 check_mc.py 的逻辑大体相同，改动的地方有两处：

1. 去掉通过 ast.parse 解析文件内容获得 tree 的步骤，直接使用参数中已经提供的 tree 变量。
2. 把打印错误信息的部分替换成 “yield (lineno, offset, msg, self)” 这样的格式，其中 self 是 Flake8 需要的返回值，在 run 方法中把 self 作为参数传入使用。

对于第二点，举个例子，之前抛出的需要添加 cache 装饰器的错误是这样的：

```
msg_p('Need `cache` decorator!')
```

现在改成：

```
yield msg_p('D012 Need `cache` decorator!')
```

msg_p 这个偏函数要改成了如下方式：

```
def msg(lineno, offset, self, msg):  
    return (lineno, offset, msg, self)
```

```
msg_p = partial(msg, body_item.lineno, body_item.col_offset, self)
```

由于集成进了 Flake8，需要在错误信息前添加错误类型，本例使用了 D012 和 D013。

接下来创建 setup.py：

```
from setuptools import setup
```

```
setup(  
    name='flake8-mc',  
    version='0.1.0',  
    ...  
    py_modules=['flake8_mc'],  
    entry_points={  
        'flake8.extension': [  
            'flake8_mc = flake8_mc:McChecker',  
        ],  
    },  
    install_requires=['flake8']  
)
```

一定要在 entry_points 中指定刚才添加的 McChecker 类。

然后安装它：

```
> cd ~/web_develop/chapter15/section2/flake8-mc  
> python setup.py install
```

现在查看 Flake8 扩展信息就可以看到 flake8-mc 了：

```
> flake8 --version  
2.5.4 (pep8: 1.7.0, flake8-mc: 0.1.0, pyflakes: 1.0.0, flake8_quotes: 0.3.0, mccabe:  
0.4.0, flake8-print: 2.0.2) CPython 2.7.6 on Linux
```

使用 Flake8 来检查：

```
> flake8 ../movie_order.py # 没有错误
> flake8 ../movie_order_wrong.py
../movie_order_wrong.py:15:5: D012 Need `cache` decorator!
../movie_order_wrong.py:24:5: D013 Need clear mc!
../movie_order_wrong.py:36:5: D013 Need clear mc!
```

代码评审的意义

Web 开发大多是团队协作的。假设实现代码的人叫小明，小明在合并代码之前需要考虑如下问题：

1. 功能实现是否符合预期，上线后会不会出现非预期的错误？
2. 代码有没有影响到本产品或者其他产品现有的其他功能？尤其是那些功能并非小明实现的功能？
3. 是否有不合理的设计，代码实现是不是使用了最好的方式，有没有性能问题？
4. 实现的代码是否有安全问题？
5. 如果小明休假甚至离职了，这部分功能出现了问题，其他人完全不熟悉怎么办？

豆瓣的合并代码有一个不成文的规定：每个 PR 都会需要有作者之外的人来做代码评审，评审者发出至少一个 LGTM（Look Good To Me 的简称）的评论、没有其他未解决的争议、测试通过，这三点要求都达到了才可以让代码合并到主干上。

这样做的好处是：

- 更容易发现一些纰漏。一个人的思考总是会不可避免地出现一些不周全的地方，而这些纰漏在别人眼中也许显而易见。
- 至少保证有另外一个人也了解这个功能的实现。
- 在代码评审中可以从其他同事身上学到很多技巧和经验。
- 促进同事之间的沟通与交流。

项目的代码质量高可以加快开发的速度，因为它会使得迭代、协作和维护更加容易。代码评审是对项目质量的基础保障，提交代码的人当然不希望被吐槽得太狠，所以在提交前会尽量保证代码的质量，并提前为可能收到的一些对需求本身和实现逻辑的提问做好准备。

事实上，要做好代码评审非常难。接下来，笔者分享一些自己的经验和理解。

作为被评审者

当工程师提交了代码，潜台词就是“我认为我写得足够好了，效果还不错”。当其他同事不断地发表评审意见时，工程师容易产生负面情绪，尤其是新进团队的人员：

1. 需求这么多，还要花费额外的精力来做或者被做代码评审，有必要吗？
2. 某些 Python 规范或者团队规范，比如一行代码不能超过 79 个字符，是不是限制得太苛刻了，不能适当放宽吗？
3. 我不熟悉之前某功能，直接拷贝代码，改一下能跑不就行了？
4. 这些意见确实有道理，但是时间太紧了，下次改吧。
5. 我并不认同 XX 说的，我写的也有道理啊，为什么要改？
6. 都是同事，何必吹毛求疵到如此境地，放过我吧！

笔者总结，其实最难的不是自己写代码，而是维护别人写的代码，在复杂的逻辑中找到某一个隐藏得很深的 bug，或者在某个（些）位置添加一些代码以实现新的功能。你需要试着按照最初实现者的思路去理解，这往往是最难的，这个过程中非常让人容易产生挫败感和不良情绪。

如果原作者仍然在职还好，有问题直接去问，但假如他已经离职，你很可能偶然会遇到下面的问题：

1. 原作者设计得太复杂，一点小的改进都要大费周章，完全掌控他的代码需要不少时间。
2. 代码性能不好。之前因为用户量和访问量太少而相安无事，现在问题突然爆发了，拖慢了整个应用甚至影响到基础设施。
3. 想要修改功能时却发现程序里充斥着各种无法理解的逻辑，改完之后莫名其妙的 bug 一个接一个。

遗憾的是，你既可能是那个接手的人，也可能是制造麻烦的人。谁都写过烂代码，笔者偶然翻看自己两年前的代码，也会觉得不忍直视，甚至发生过看了一段代码觉得相当烂，使用 git blame 命令想吐槽原作者，却发现这是自己写的代码这样的尴尬事。

只要不是一个人完成项目就会涉及团队协作，每个人都希望别人写的代码自己能接受，这就需要遵循一些规范，代码评审就是规范代码的过程。其次，代码评审通过评审者从旁观者的角度，小到 SQL 语法，中到程序设计，大到产品方向都给予及时的意见，用更小的成本降低上线后出现问题的概率。

以笔者刚入职豆瓣提的评审意见最多的一个 PR 为例来说明（如图 15.1 所示）。

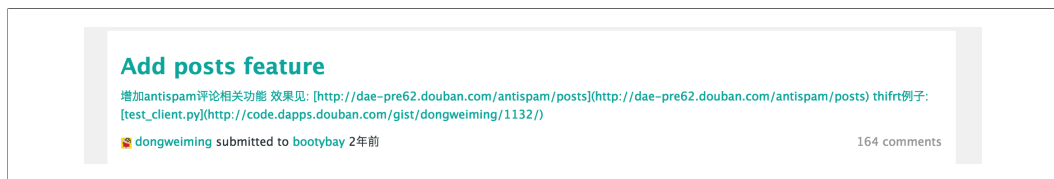


图 15.1 评审意见最多的一个 PR

发表的评审意见数量为 164。被评审者怎么摆正自己的心态呢？

一开始会被这种严格评审搞得很狼狈，但是当你习惯这种流程并且践行之后，你就是它的受益者。比如之前的那几点，笔者是这么看的：

1. 代码评审是代码合并之前不可或缺的一步，代码是团队一起维护的，大家都有责任保证整体质量。
2. 每个人都能遵守，我为什么不可以？
3. 我经常能看到大量的重复代码，它们的意图和语句很像，长此以往项目会越来越臃肿，每次修改对应的内容都需要把雷同的代码全部修改一遍，如果不能理清这些逻辑，盲目地整体替换，非常有可能造成漏改或者错改而导致上线后出现问题。
4. 大量例子证明，当场不解决，大部分就都没有后文了。现在的事情，现在做。
5. 世界上有太多的事情不是太阳东升西落这样的客观真理，也没有绝对的压倒性的意见。我们组探索的方案是投票制，少数服从多数。你不一定认同，只需要在团队项目中遵守规则即可。
6. 这不是吹毛求疵，而是精益求精，力求完美，你同样可以合理地要求其他人这么做。

作为评审者

作为评审者，笔者也有一些经验和大家分享。

1. 尊重他人，就事论事，对事不对人。可以在适当的场景下发一些 emoji 图片、语气词缓解气氛。
2. 首先要了解产生此 PR 的一些原因、需求说明等，如果你有强烈的反对意见，多花几分钟审视一下自己的意见再做反馈。
3. 询问，不要指点。比如可以说“为什么你要……”，而避免说“不要……”。
4. 解释你提出修改意见的原因。

5. 评论旨在提高专业技能、产品质量、达成团队共识。很多代码风格或者用法确实是因为个人习惯这么写，大家都可以接受就行了。如果确实觉得需要改善，评论之前可以按以下顺序找到最佳用法的示例再去评论。

- Python 标准库中的用法，最好以 Python 3.5 的风格为标准。
- 知名开源项目中的用法。
- 其他项目中的用法。

人工反馈很可能会由于一时无法控制情绪而用词不当，所以尽量让程序来反馈。为此才引入 mention-bot、给 CI 添加 Flake8 检查、使用 Gandalf 以及其他保证产品代码质量的工具。

评审者为了让大家接受自己的意见，必然需要理解需求并找到支持自己观点的各种佐证，这个过程既帮助深入理解了产品，也能提高自己的技术水平。

评审的标准

前面说的都是烂代码，不好的习惯。那么，好的代码是什么样子呢？

Bjarne Stroustrup（C++ 之父）说：

- 逻辑应该是清晰的，bug 难以隐藏。
- 依赖最少，易于维护。
- 错误处理完全根据一个明确的策略。
- 性能接近最佳，避免代码混乱和无原则的优化。
- 整洁的代码只做一件事。

Michael Feathers（《修改代码的艺术》作者）说：

- 整洁的代码看起来总是像很在乎代码质量的人写的。
- 没有明显的需要改善的地方。
- 代码的作者似乎考虑到了所有的事情。

可以感受到，对好的代码的理解有很多共通的地方：

- 代码简单。你的代码意图明确，其他人才容易与你协作。
- 可读性和可维护性要高。
- 以最合适的方式解决问题。

好的 PR 是什么样子呢？

1. 标题要涵盖此 PR 的目标，比如：修复 XXX 问题，优化 XXX 问题，增加 XXX 特性。如果做的修改涉及多个特性，可以使用列表挨个列出来。
2. 提供此 PR 起因的说明（比如 Trello 相关链接等），不要假定对方已经熟悉这些事情的前因后果。这样做也是为了在未来别人追溯起来的时候能理解这件事。
3. 此 PR 的主体应该包含一个可测试的地址，可以让评审者预览效果。
4. PR 中的每一个 commit 日志都应该可以和代码对应，方便评审。
5. 尽量不要发太大的 PR，小步迭代，把整个工程拆分，不断向主干提交。
6. 提 PR 之前已经确保在本地或测试环境一切正常。
7. 写好的 commit 信息，禁止使用“fix bug”、“commit”这样无意义的 commit 信息污染 commit 树。社区有多种写法规范。推荐参考写出好的 commit message（<http://bit.ly/1XYn2Y6>）这篇文章。另外推荐 Thoughtbot 的 commit 信息模板，使用方式如下：

- 创建 ~/.gitmessage，输入：

```
# 50-character subject line
#
# 72-character wrapped longer description. This should answer:
#
# * Why was this change necessary?
# * How does it address the problem?
# * Are there any side effects?
#
# Include a link to the ticket, if any.
```

- 在 Git 全局配置文件 ~/.gitconfig 中添加：

```
[commit]
    template = ~/.gitmessage
```

以后每次执行 git commit 的时候就会时刻提醒你遵循标准了。

除此之外，被评审者应该非常熟悉 Git 的使用，如 rebase、cherry-pick、reset --soft 等命令。